

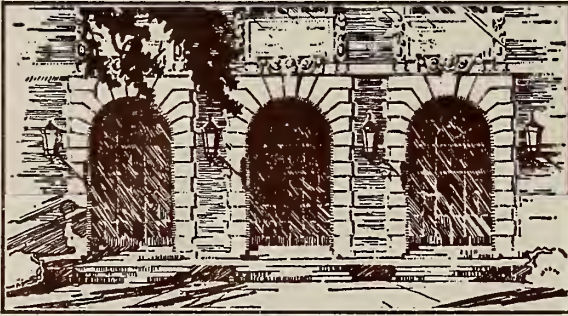
LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84

I26r

no. 535-540

cop. 2



CENTRAL CIRCULATION BOOKSTACKS

The person charging this material is responsible for its renewal or its return to the library from which it was borrowed on or before the **Latest Date** stamped below. **You may be charged a minimum fee of \$75.00 for each lost book.**

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

TO RENEW CALL TELEPHONE CENTER, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

SEP 16 1996

When renewing by phone, write new due date below
previous due date.

L162



Digitized by the Internet Archive
in 2013

<http://archive.org/details/designimplementa536mill>

10.84
llbr
0.536
op. 2

Math

UIUCDCS-R-72-536

THE DESIGN AND IMPLEMENTATION OF A MULTI-TASK
BATCH OPERATING SYSTEM AND PAGING
INTERPRETER FOR THE PDP-11

by

James Douglas Miller

THE LIBRARY OF THE

August, 1972

SEP 12 1972

UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-72-536

THE DESIGN AND IMPLEMENTATION OF A MULTI-TASK
BATCH OPERATING SYSTEM AND PAGING
INTERPRETER FOR THE PDP-11

by

James Douglas Miller

August, 1972

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

*This work was supported in part by the National Science Foundation under Grant No. US NSF GJ-812 and was submitted in partial fulfillment for the Master of Science degree in Computer Science, 1972.

ACKNOWLEDGEMENT

The author would like to thank Professor Donald B. Gillies for his advice and encouragement throughout this project. I am also indebted to Professor George Friedman for reading the completed manuscript in the absence of Dr. Gillies. Earle Heffley deserves special recognition for his dedication to the design and development of the MIPS interpreter. Many details of monitor design evolved from discussions between Earle and the author.

I am indebted to Don Oxley for many ideas and suggestions, especially in the areas of disk management and priority scheduling. Special thanks also go to Jim Hart for his work on the PAL-11A ETS assembler, and to Rick Cattell who is modifying that assembler for installation into MIPS. I would also like to thank Al Davis and Ian Stocks for reading and commenting on the manuscript, Bob Miller and Harold Lopeman for relieving me of some of my hardware duties, and Mark Ketelsen, my officemate, for suggesting many acronyms for the system, none of which were suitable for a published report.

The author is very grateful to Mrs. June Wingler for her excellent typing and for her patience during the writing of this paper. Without her attention to detail, the text would never have cleared format checks. Finally I am indebted to my wife, Gen, for still being there when I immersed from my six month hibernation in the basement of the computer lab.

TABLE OF CONTENTS

	Page
I. INTRODUCTION.....	1
II. BLOCKS, BUFFERS, QUEUES, and CODE.....	3
A. Memory Partitions.....	4
B. System Generation.....	4
C. System Buffers and Blocks.....	5
D. Job Control Blocks.....	7
E. Non-Resident System Code.....	11
III. DISK MANAGEMENT.....	13
A. Skewed Tracks.....	13
B. Disk Queue and Service Routines.....	17
C. Optimization Strategies.....	19
IV. MIPS FILING SYSTEM.....	23
A. MIPS File Structure.....	23
1. File Windows.....	25
B. Types of Files.....	26
C. User File Directory.....	27
D. File Service Requests.....	28
1. .CREATE.....	31
2. .OPEN and .OPENE.....	31
3. .READ and .WRITE.....	33
4. .RLSE.....	34
5. .MARKF.....	35
6. .CLOSE.....	35

7.	.DELETE.....	36
8.	.WAITZ.....	36
V.	MIPS STRUCTURE AND OPERATION.....	38
A.	System Tasks.....	38
1.	Card Reader Service Routines.....	38
2.	Line Printer Service Routines.....	40
3.	The MIPS Swapper.....	42
4.	Console Service Routines and System Communication.....	46
B.	User Tasks.....	49
1.	FILE Tasks.....	51
2.	EXEC Tasks.....	51
C.	Task Management.....	52
1.	System Priorities.....	52
2.	The MIPS Dispatcher.....	53
3.	Sequencing between Tasks of a Job.....	56
VI.	SYSTEM PROGRAMS.....	58
A.	The Assembler.....	58
B.	The MIPS Loader.....	58
VII.	A VIRTUAL MEMORY INTERPRETER.....	60
A.	Paging the User Program.....	61
1.	Paging When Core is Not Full.....	61
2.	Least Recently Used Replacement Policy.....	64
3.	Floating Interpreter Pages.....	65
B.	The Virtual Machine.....	66
1.	Memory Configuration.....	66
2.	Instruction Set.....	66
3.	Error Traps.....	67

C. Access to User Files.....	69
1. Calls to the MIPS Filing System.....	69
2. Record Oriented I/O.....	70
D. Diagnostic Tools.....	72
1. Dump Routines.....	72
2. Trace Routines.....	73
E. Interpreter JCL and Options.....	74
1. Object File Name.....	76
2. Processor Error Handling Options.....	76
3. File Error Handling Options.....	76
4. Instruction Count.....	77
5. Line Count.....	77
6. Control Card Format Variations.....	77
II. CONCLUSIONS.....	78
LIST OF REFERENCES.....	80
APPENDIX	
A. FILING SYSTEM ERROR CODES AND PROGRAMMING EXAMPLE.....	81
B. CONTROL CARDS AND DECK SETUP.....	86

LIST OF FIGURES

	Page
1. System Generation Procedure.....	6
2. Job Control Blocks.....	9
3. PAL Call Format for Non-resident Routines.....	12
4. Disk Layout.....	16
5. Disk Parameter Block.....	18
6. File Structure.....	24
7. PAL Formats for File EMTs.....	29
8. File Information Block Field Definitions.....	30
9. Swap Example.....	45
10. Sample Log and Command Strings.....	50
11. Dispatcher Flowchart.....	55
12. Interpreter Tables.....	63
13. Interpretation of Double Operand Instructions.....	68
14. Call Formats for Interpreter Defined EMTs.....	75

I. INTRODUCTION

This paper is primarily concerned with the design and implementation of a batch operating system for the PDP-11. The system is called the Multi-task Interpretive Paging System or MIPS to reflect the fact that user programs run under an interpreter which pages the user code in order to implement a virtual machine. MIPS was designed to serve a community of inexperienced student programmers taking an introductory course in assembly language programming. This application requires fast turnaround for small student jobs without precluding simultaneous use by system programmers and research personnel who may want to run jobs of several hour duration.

MIPS is intended for use on an existing target machine with a particular hardware configuration. The system includes a PDP-11/20 processor with 20K words of core memory, a 256K word fixed head per track disk, a dual DECTape transport, several terminals including a console teletype, a 600 cpm card reader, and a 500 lpm line printer. Operating system development is somewhat complicated by the fact that the PDP-11/20 does not provide any form of memory protection, segmentation, or relocation hardware. MIPS is a batch system because, in the opinion of the author, this configuration is not well suited to timesharing due to inadequate disk capacity. The monitor therefore does not attempt to support remote access from terminals.

In the MIPS multi-task environment, system tasks handle spooling of cards to disk and listings to the printer while user tasks run concurrently in two fixed partitions. To prevent long jobs from tying up the CPU, user

tasks running in the larger of the two partitions are swapped out to disk after a preset time interval and are given additional time slices in round robin competition with other jobs in the system. MIPS supports a job mix of up to six jobs and manages all resources required for execution of these jobs. Disk management routines attempt to optimize disk access time to improve system performance. In general, fairly conventional multiprogramming techniques are used throughout the system design. All monitor code was written by the author and consists of some 4300 cards. It is a modular system to permit simple restructuring of partitions. To promote understanding, monitor code is extensively commented and avoids "clever" coding practices.

The design and implementation of MIPS is described in the subsequent chapters. The text does assume that the reader is familiar with the PDP-11 but the discussions are not so detailed that experience with the PAL assembly language is required. We begin in Chapter II with definitions of several terms and descriptions of several structures. Chapter III discusses the disk management and optimization strategies used in MIPS and Chapter IV describes the filing system. We defer discussion of actual system operation until Chapter V since the previous topics are essential to an understanding of system tasks, user tasks, and task management routines. Chapter VI contains brief comments on system programs including the assembler and loader. Chapter VII, dedicated to the MIPS interpreter, details the implementation of software paging mechanisms necessary to implement a 32K virtual PDP-11. Finally we draw a few conclusions in Chapter VIII.

II. BLOCKS, BUFFERS, QUEUES, and CODE

As the title may suggest, this chapter includes the definitions of several global terms and data structures. The chapter is essentially a collection of almost unrelated topics which have no optimal placement within the text but are best introduced near the beginning.

The terms task and job are used throughout this chapter and the rest of the paper so a formal definition is in order. We permit the term task to have a very general definition and use it to refer to a particular system module, i.e., the code itself, or to refer the execution of a set of instructions to achieve some end. Thus, the collection of routines which handle reading of cards is identified as the system task CARD. In another context, a user task will refer to the execution of a program such as an assembler. A job is more clearly defined as a set of user tasks, or, in a physical sense, as a deck of cards delimited by a \$JOB card and a \$END card.

We will also refer to several queues throughout the text. With only the two exceptions noted below, all queues in MIPS are simple doubly linked lists. There are six system queues used by job and task management routines. The disk routines manage a queue of disk requests. We will also refer to message and log queues. These latter two types of queues are implemented as singly linked lists.

Since MIPS is a fixed partition system, we begin with a brief discussion of those partitions. The section on system generation will be important to whoever takes over system development and will serve to introduce the reader

to the disk layout discussed in detail in Chapter III. System buffers, system blocks, and Job Control Blocks are global data structures referenced throughout the text. Finally, we mention non-resident system code basically because it fits in with the discussion of system generation and system blocks.

A. Memory Partitions

The core memory configuration on the target system was upgraded from 12K to 20K words during MIPS development. The eventual 20K configuration was uncertain, however, so MIPS was planned with a 16K system in mind. Memory was initially divided into a 6K system area, a 2K FILE partition, and an 8K EXEC partition. MIPS outgrew its 6K area, however, so the system was reconfigured for an 8K system area, a 4K FILE partition, and an 8K EXEC partition when the 20K configuration became firm. Since MIPS is a very modular system, rearrangement of partitions requires little more than a reassembly. This latter allocation was considered reasonable for the initial release. If major surgery were performed, the system area could conceivably be reduced to 4K. This would allow simultaneous operation of one, two, or three 8K user partitions on a 12K, 20K, or 28K machine. This more general approach was avoided because it adds considerable complexity to routines such as the dispatcher and requires that all system routines be at least statically relocatable.

B. System Generation

When the system area expanded to 8K, a non-resident code facility was added to permit even further expansion. Non-resident routines are assembled along with the rest of the system starting at address 40000_8 , i.e., above the resident system code. Symbol table space was inadequate for full system assemblies on the 12K PDP-11 system. Assemblies were run on the University's

360/75 system using a PAL cross assembler. Included in the non-resident routines is a small "bootout" program used to generate the disk copy of the system. Core is merely loaded from the 360 object deck and the whole package is written out to disk. The disk copy, mapped in figure 1, includes a system boot used to reload the system from disk, a copy of the resident system, and all non-resident routines. The copy of the resident system is kept on disk as a convenience but is not really required. To prevent the disk allocation routines from overwriting these system blocks with user files, appropriate bits in the disk bit maps are cleared at assembly time. The system generation procedure described here will be adequate for up to 48 non-resident segments if system core totals 20K words.

C. System Buffers and Blocks

As planning for the system progressed, it became apparent that some type of system buffer pool would be required to satisfy the many transient requirements for small blocks of core. The current MIPS implementation requires 13 word blocks to serve as User File Directory (UFD) entries, and ten word blocks for use by the disk service routines and filing system. Temporary storage of up to 72 characters is also required for system messages but a single message may extend over several system buffers. To satisfy these requirements MIPS maintains a pool of 100 buffers, each containing 14 words. The pool is implemented as a singly linked list. Buffer allocation and return mechanisms are simple system subroutines. If a buffer is available at the time of a request, the allocator will delink a buffer from the free chain, clear it to zeroes, and pass the buffer address to the calling routine. If a buffer is not available, the job is put to sleep and remains dormant in core until a buffer is returned to the pool by another task. Since buffers may be allocated to long term, i.e., life of a job, applications, there is

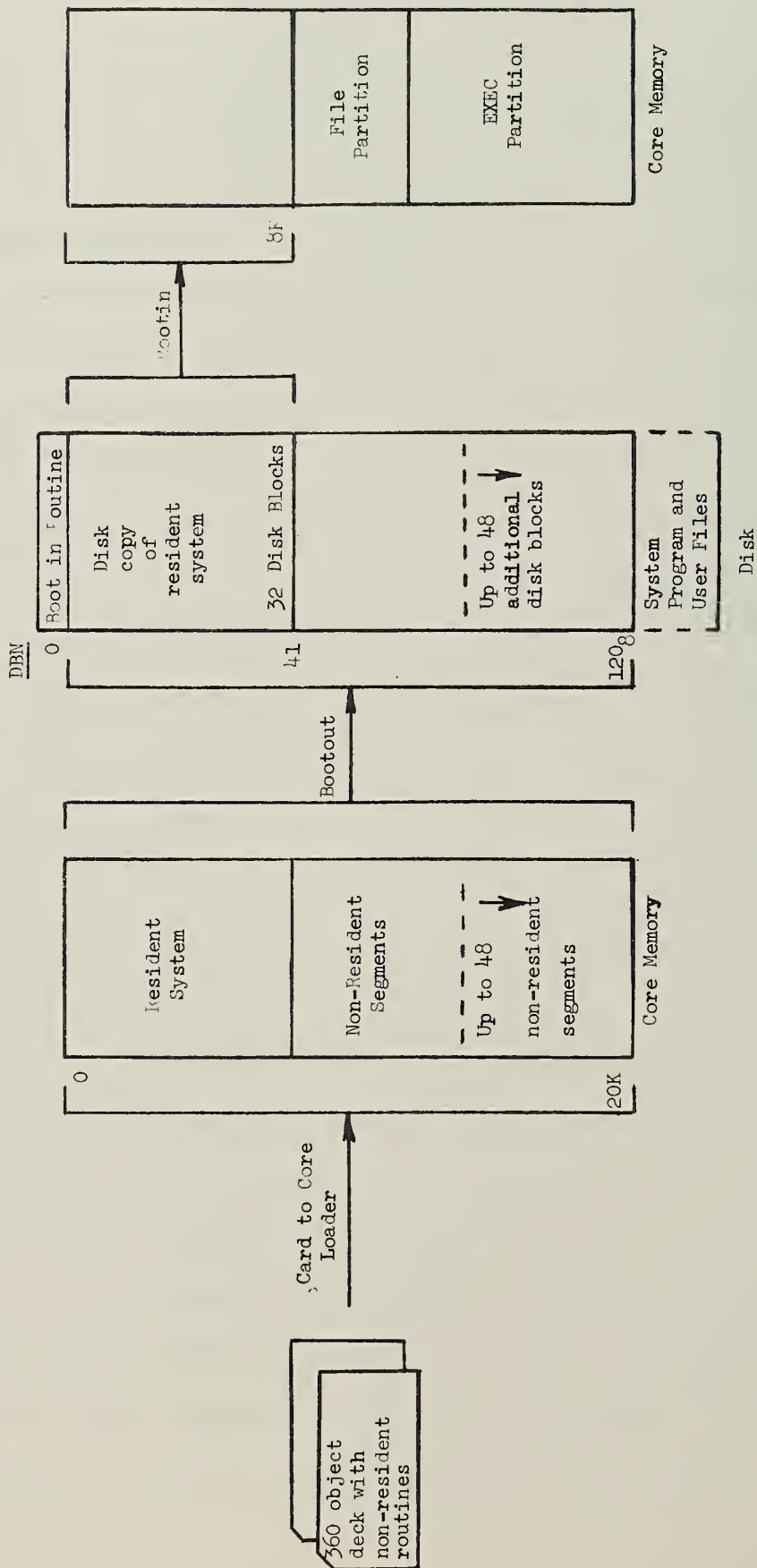


Figure 1. System Generation Procedure

a possibility of system deadlock due to running out of buffers. The pool is large enough that this possibility is very remote and deadlock prevention or even detection code is not considered worthwhile for this application.

Dynamic allocation of larger blocks of core is required for loading non-resident code segments and to allow fast deletion of files. MIPS provides two 256 word "system blocks" for these applications. Block allocation and return routines are similar to those used for system buffers. The block allocator will also put a job to sleep if a system block is not available. However, since blocks are never allocated for more than a few hundred milliseconds and MIPS routines are careful not to require more than one at a time for any task, there is no possibility of deadlock. The small number of system blocks does impose one restriction on non-resident code segments which is discussed in section E.

D. Job Control Blocks

A Job Control Block (JCB) is a 64 word section of core memory which is assigned to a job by the system task CARD immediately after the \$JOB card is read and remains with the job throughout its life. MIPS maintains a pool of six JCBs. Six jobs may therefore be in the system at any one time. Initially all JCBs are linked to a card reader queue for assignment to incoming jobs. Once assigned to a job, the JCB remains on top of the reader queue until the deck has been read. When the \$END card is encountered in the input stream, the job is initialized by delinking its JCB from the reader queue and linking it to the EXEC queue to await execution. Since tasks of a job may run in either the FILE or EXEC partition, the JCB will be linked to either the FILE or EXEC queue as required during execution. After all processing has completed, the JCB is added to the print queue and the system task LINE handles printing

of all listable output files. Finally, at end of job LINE clears the JCB and returns it to the reader queue for reassignment. In subsequent sections of this paper we will often equate the terms "job" and JCB and refer to a job being linked to a queue.

The JCB is used in some way by nearly every MIPS routine. To point out where the various fields are used, a brief commentary is included below. More detailed discussions are deferred to subsequent sections of this paper. Field names and positions are shown in figure 2. Blank positions in the figure denote words which are available for future assignment.

The first 16 words of the JCB include the job identification number (JCBID), all important status indicators (JCBST1, JCBST2, JCBWAT, and JCBIO) required by the scheduler and swapper, and a register save area used whenever the job is put to sleep. The mix index bits in the JCBID word are actually the JCB number (1-6). These three bits are never changed so the particular JCB assigned to a job determines the job's mix index. The disk claim (JCBCLM) is a \$JOB card parameter. It is used by the filing system in conjunction with the count of disk blocks used (JCBUSD) to prevent a job from usurping more disk space than was claimed. JCBMAX, JCBNOC, and JCBNOL are merely job statistics which are printed at end of job. JCBMSG is the list head for a queue of messages en route from the console TTY to the job. Since the JCB is always resident, messages may be sent to the job, i.e., added to the queue, at any time even though the job may not be in core. The system return code (JCBRTN), printed at end of job, relates the reason for termination if the system aborts the job. JCBPRC is used by the EXIT routine to hold the name of the next task during sequencing between tasks. The MIPS swapper scans the five word Real Memory Table (JCBRML-JCBRM5) to determine which blocks are in use. Finally the last 14 words of the JCB comprise seven file slots which contain filing system pointers for open files.

JCBRLK	Right link
JCBLIK	Left link
JCBID	Bits [15-12] not used; [11-9] mix index; [8-0] user Id number
JCBST1	Current job status
JCBST2	Job status save
JCBWAT	Address of wait marker
JCBRO	R0 Register save area
JCBR1	R1
JCBR2	R2
JCBR3	R3
JCBR4	R4
JCBR5	R5
JCBR6 or JCBSP	R6 Stack pointer
JCBR7 or JCBPC	R7 Program counter
JCBPS	Processor status word
JCBIO	System marker for all disk I/O for job
JCBCLM	Number of disk blocks claimed by job
JCBUSD	Number of disk blocks currently in use
JCBMAX	Maximum number of disk blocks used by job
JCBPRC	Procedure name storage during task sequencing
4th char 3rd char	2nd two characters of procedure name
6th char 5th char	3rd two characters of procedure name
JCBMSG (RLINK)	Message queue list head - right link
(LLINK)	Left link - currently not used
JCBNOC	Number of cards read
JCBNOL	Number of lines printed
JCBRTN	System return code

Continues on
next page

Figure 2 . Job Control Block

100		
102		
104		
106		
110		
112		
114		
116		
120		
122		
124		
126		
130		
132	JCBRM1	Real memory table (RMT) 040000-057777 8K-12K
134	JCBRM2	RMT for core addresses 060000-077777 12K-16K
136	JCBRM3	RMT for core addresses 100000-117777 16K-20K
140	JCBRM4	RMT for core addresses 120000-137777 20K-24K
142	JCBRM5	RMT for core addresses 140000-157777 24K-28K
144	JCBCDR	File slot for pseudo reader file. Same as $\triangle 1$
146	Window address	Same as $\triangle 2$
150	JCBLPT	File slot for pseudo printer file. Same as $\triangle 1$
152	Window address	Same as $\triangle 2$
154	JCBSWP	File slot for swap file. Same as $\triangle 1$
156	Window address	Same as $\triangle 2$
160	JCBFIL - Slot 1	Floating file slots for user files. UFD address.
162	Window address	Same as $\triangle 2$
164	FILE - Slot 2	$\triangle 1$ Address of UFD entry for file.
166	Window address	$\triangle 2$ Address of system buffer which holds window.
170	FILE - Slot 3	Same as $\triangle 1$
172	Window address	Same as $\triangle 2$
174	FILE - Slot 4	Same as $\triangle 1$
176	Window address	Same as $\triangle 2$

Figure 2. Job Control Block (continued)

E. Non-Resident System Code

In the initial design of MIPS, the unfortunate and somewhat naive assumption was made that a core resident system could be written to fit into a 6K system area. Consequently, the need for a non-resident (NR) code facility was not recognized until late in development when the system expanded beyond its intended limits. Out of necessity the necessary code to locate and load a non-resident code segment was implemented and has proven very useful in several areas. Non-resident routines are currently used to generate burst printouts, to set the system clock, and as response routines in the console command string interpreter. Several currently resident routines, e.g., the swapper, are also marked for conversion to non-resident segments.

Since non-resident routines are loaded into system blocks for execution, the code must be position independent and blocked in 256 word segments. More than one routine may be in a segment but none may cross a 256 word boundary. With the NR code segments on disk as diagramed in figure 1, actual loading of a non-resident routine is a simple matter handled by an EMT routine called .NCODE. The call has the format shown in figure 3. Since the assembly address of an NR segment determines its disk address, a routine may be referenced by name, i.e., label or address. .NCODE extracts from the assembly address a Disk Block Number for the segment and a start offset of the desired routine from the beginning of its segment. .NCODE then requests a system block and loads the addressed segment into the assigned block. The start offset is added to the address of the assigned block and execution begins at this relocated address. On completion the NR routine returns to .NCODE which will return the system block to the available pool and return to the caller. Parameters may be passed to or from non-resident routines in any of the general registers. Parameters may also be passed on the caller's stack or as additional EMT parameters if the details of the .NCODE routine are understood.

.NCODE	; EMT call
.WORD Label	; Assembly label for routine called

Figure 3. PAL Call Format for Non-resident Routines

The current implementation does not permit nesting of non-resident routines, i.e., NR routines may not call other NR routines. This restriction is imposed because an additional system block would be required for each NR routine called. This restriction may be lifted if additional system blocks are made available. A better implementation might permit one non-resident segment to be overlayed by another segment but eventually restored. To avoid saving the resident copy of a segment to be overlayed, non-resident segments would be implemented in pure code. Nesting could conceivably continue to any level and recursion would be permitted.

III. DISK MANAGEMENT

The development of a disk management philosophy was of primary importance in the initial design of the MIPS system. A traditional approach to disk management for small systems has been to segment the disk into fixed length blocks and to use either a contiguous or linked (or both) allocation scheme. Allocation of contiguous disk space for files or swap areas in multi-user systems either wastes disk space or leads to fragmentation and the use of garbage collection routines to clean things up periodically. The major objection to linked allocation schemes is that access is sequential and therefore slow. Although processing is usually overlapped with disk transfers, few systems attempt to optimize disk access time or service disk requests with anything other than first in first out queuing routines. A design goal in MIPS was to develop a technique in which contiguous allocation of disk space and reservation of large swap areas would not be required. The approach taken and described below permits a single management philosophy for the whole disk, allows blocks of files or object code to be scattered randomly over the disk, avoids most of the problems of linked organizations, and permits considerable optimization of disk access time.

A. Skewed Tracks

The only backing store available on the PDP-11 used during development was a single platter (expandable to eight platters) fixed head per track device with a capacity of 512K characters, average access time of 17 milliseconds, and DMA transfer rate of 125,000 characters per second. In any multi-programmed

environment 512K characters is very limited capacity and hence disk space was a precious resource. The platter consists of 128 tracks of 2048 words addressable to the word. MIPS treats the disk as a paging drum where pages are fixed length blocks of 256 words or 512 characters. This block size was chosen for programming convenience, as will be apparent in the discussion which follows, and as a happy compromise between a small block size which would increase disk traffic to unreasonable levels, and a large block size which would require a lot of core for I/O buffers. Eight blocks fit nicely on a track for a total of 1024 blocks for the platter.

A skewed track layout was used to help optimize disk access time. This layout is best explained with the help of figure 4a which is intended to represent one full platter of 1024 blocks. In this linearization the 128 heads sit on the YZ surface and disk rotation is right to left. Instead of referring to 128 tracks, figure 4a refers to eight planes each consisting of sixteen tracks. To clarify things the track (00-17₈), block (0-7), and plane (0-7) bits map into a disk block number (0000-1777₈) as shown in figure 4b. Similarly, the mapping into the actual disk address is shown in figure 4c. Platter bits are included in figures 4b, c, and d to suggest extension to additional platters.

Surface XZ, shown with blocks numbered, represents track 0 across all eight planes, i.e., disk block numbers 0000, 0001, ..., 0077. Notice that the blocks on plane 1 are indented or skewed by 32 words from the corresponding block on plane 0. Similarly, plane 2 is skewed 32 words from plane 1, etc. The skewing provides 64 discrete time frames, spaced 512 microseconds apart, for consideration by the optimization routines. It is important to realize that at some time say t_{14} , i.e., when the first word of block 0014 is in position under the heads, the first words of DBNs 0114, 0214, ..., 1714 are also under the heads.

One complication has been omitted from the introduction above. Consider a disk transfer on DBN 0072, i.e., block 72 on the XY surface in figure 4a, and note that track switching actually occurs 64 words before the end of the block. Since this particular disk has spiral read/write capability, the 64 words are merely read from track 1. Since track 1 on plane 2 is also skewed by 64 words, DBN 0072 simply bridges two adjacent tracks. This spiral can continue across all 16 tracks of plane 0 up to DBN 1772. Here a similar thing occurs and the 64 words are read from track 0 on plane 3. However, plane 3 is skewed by 96 words so there is a gap of 32 words between the end of DBN 1772 and the beginning of DBN 0003. In fact, gaps exist between all adjacent planes. These seven gaps, totaling $7 \times 32 = 224$ words, are simply dead space which cannot be accessed as a consequence of the skewing scheme.

An interesting consequence of the skewing is the phenomenon of DBN 1777. Track switching will occur 32 words from the beginning of this block and there is no place to pick up the remaining 224 words. Even if an additional platter existed, the spiral could not continue to that platter since it would be identical to the platter diagrammed in figure 4a, i.e., track 0 would not be skewed. At first glance it appears that block 1777 is also lost to dead space in addition to the 32 word gaps mentioned above. In reality, the 32 word gaps are merely absorbed by this end block. 224 words have been lost and they are all in DBN 1777. The problem is solved by permanently allocating that block in the bit maps. The disk is therefore reduced to 1023 available blocks.

The mapping of blocks into bit maps as shown in figure 4d is a very natural consequence of this organization. Each of the 64 time frames (or blocks on the XY surface of figure 4a) is represented by one word in the bit map and each of the sixteen blocks in the frame is represented by one bit in

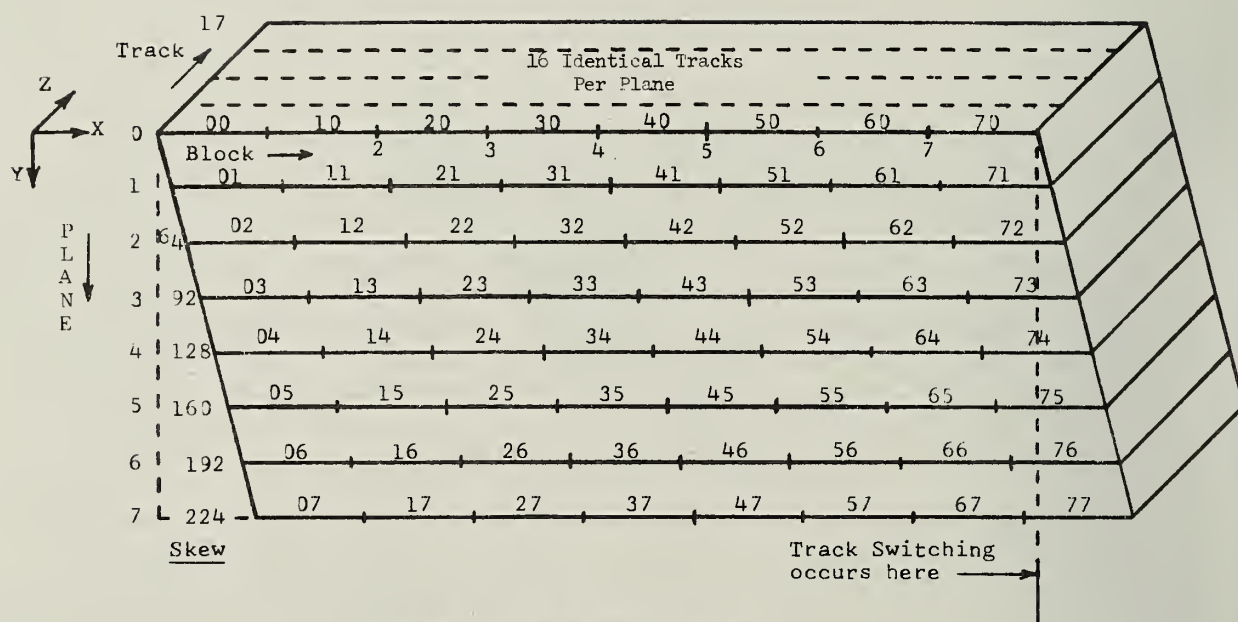


Figure 4a. One Disk Platter

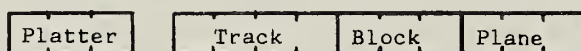


Figure 4b. Disk Block Number Format

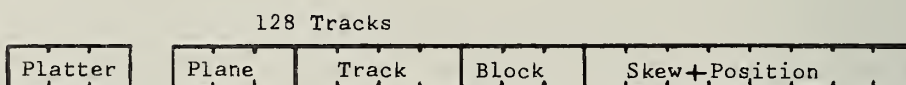


Figure 4c. Disk Address

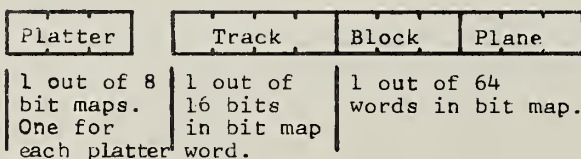


Figure 4d. Bit Map Correspondence

Figure 4. Disk Layout

the bit map word. Extension to several platters is accomplished simply by adding additional bit maps.

B. Disk Queue and Service Routines

All disk requests, except those for non-resident system code, originate in the filing system. To request disk access a disk parameter block is constructed in a system buffer allocated from the buffer pool. The buffer address is then passed to a queuing routine which links the request to a disk queue. The queue was implemented as a doubly linked list of disk parameter blocks. A parameter block, diagrammed in figure 5, consists of two link fields and seven words of information. Five extra words are available for possible future use. The first parameter specifies the disk block number (DBN) and type of transfer (read or write). If the request references a specific disk block, as will be the case for all read requests and some partial block writes, the DBN will not be zero. However, in the case of full block writes, there is good reason to zero the DBN and let the disk service routines select a DBN. The filing system does not care where the block is written as long as the DBN is returned on completion of the transfer. The core memory address parameter is merely the address of a source or destination buffer in core. Position and size fields allow partial block transfers and are used by card reader and line printer service routines to obtain double buffering within a single 256 word core buffer and by the filing system to gain access to sections of File Control Blocks. The last four words are marker addresses used for posting completion of the transfer. Markers 1, 2, and 3 are essentially semaphores which are incremented when the transfer request is queued and decremented upon completion. On write/allocate transfers, the DBN is returned to the address of marker 4.

Right Link			
Left Link			
1 R/W	Not used	3 Disk Block Number	10
Core Memory Address			
START WORD(Position) 0-255		SIZE 0 = 256 Words 377 = 255 Words	
Address of Marker 1			
Address of Marker 2			
Address of Marker 3			
Address of Marker 4			
Last 5 words of system buffer are currently not used			

Figure 5. Disk Parameter Block

The disk service routines includes a queuing routine, an interrupt service and dequeuing routine, and the allocation and deallocation routines. The queuer is responsible for linking parameter blocks into the disk queue, pre-transfer marking, and initialization of certain transfers. The interrupt service routine posts completion of transfers, delinks requests from the queue, and is responsible for queue scanning and initialization of requests not handled by the queuer. The allocator and deallocator manage the disk bit maps. The allocator uses a two level search technique to reduce the worse case allocation time. The first level map consists of 64 bits describing the status of the 64 words in the second level map. On the lower level is the primary bit map which indicates status (in use or free) of each of the 1023 disk blocks. The disk organization described in the previous section and the two level allocator reduces the average time to allocate a block to less than 100 usecs. even if the disk is 75% full. Worse case allocation time, i.e., the time required to allocate the last free block under the worse possible conditions, is only 450 usecs. The queuer, interrupt service routine, and allocator work together to optimize disk access time as outlined in the next section.

C. Optimization Strategies

The disk organization described lends itself to several optimization strategies provided that the disk position can be known at any point in time. Head per track devices commonly provide a hardware register which can be read by the processor to determine disk position. In this case an 11 bit disk segment register (DSR) could be interrogated to determine disk position to any resolution desired (down to a word). If we throw away the lower 5 bits of the DSR, we get position to 32 word resolution and this corresponds to the 64 time frames mentioned above.

To begin the discussion of disk access optimization, consider a request to write one block out to disk. For the moment, assume the disk queue is initially empty and the block to be written is newly created information, i.e., it does not replace any block already on disk. Upon entering the queuing routine with disk parameter block in hand, the queuer immediately determines that the disk queue is empty and the new request is to be initialized. However, a disk block has not as yet been allocated ($DBN = 0$) for this information. Before calling the allocator the queuer will read the DSR. For clarity, assume the DSR reading is 45₈, i.e., the disk is somewhere within the first 32 words of time frame 45. Since some time is actually required to examine queues, initialize a transfer, and to allow the disk electronics to settle, a timing margin of 3 is added to the DSR reading giving 50 as the next available time frame. The queuer will therefore request a block from the allocator starting with frame 50 and wrapping around to the right in figure 4a to frame 47, i.e., the range of acceptable blocks spans the whole disk. The allocator responds by trying to find a block in frame 50 or, failing that, moves on to frame 51, 52, ..., 00, 01, ..., 47 until a block is found.

An important point to make here is that there are sixteen choices for blocks in time frame 50. Assuming the disk is half full, i.e., 512 blocks are in use and randomly distributed over the surface, the probability of finding at least one free block in time frame 50 is between .99 and 1. In fact the distribution is very flat and the probability of finding a block in the desired time frame does not drop below .5 until the disk is 96% full. The average access time for write/allocate transfers is, therefore, less than 1.5 milliseconds instead of 17 milliseconds as would be the case in conventional schemes. In this example, if the new block had replaced an existing block on disk, access would be much better if the old block were deallocated and the new block written into a different disk block.

MIPS employs an extension of the previous concepts to improve access time when the disk queue is not empty. The simplest case involves the queuer confronted with a new request and a non-empty queue. In this context the disk controller will have been conditioned to perform one of the requests on the queue. However, a pending transfer can be aborted simply by clearing the controller registers and it will serve our purposes to do so occasionally. The word "pending" should be emphasized in this regard. A transfer is considered pending until the disk rotates into position to begin moving words to or from memory. It would be senseless to abort a transfer which has actually begun.

In this situation the queuer examines the DBN of the pending transfer and computes a range of time frames which will give better access time than the pending transfer. If the DBN of the new request is within this range or if a block can be allocated in the acceptable range, the pending transfer is aborted and the new request initialized. An aborted transfer merely remains on the queue for service at a later time.

This is a good place to bring up one final complication. Since blocks overlap, permanent or semi-permanent blocking of a request can result if these routines are not careful. To avoid this, the range of acceptable blocks computed above depends on whether or not the pending transfer is on top of the queue. Aborting the top request is permitted only if the new request can be completed without delaying that request in any way. Transfers not on the top of the queue can be arbitrarily preempted since they will quickly bubble up to the top.

The disk interrupt routine gets into the optimization game at the completion of transfers, i.e., when a disk interrupt occurs. If the queue is not empty, a new transfer must be initialized. Under the same discipline

imposed above, the request on top of the queue is given priority over lower requests and a range of acceptable frames is computed. The next eight requests on the queue are then scanned in an attempt to find the best transfer (least access time based on current disk position) which can be performed without delaying the request on top of the queue.

The description of the optimization strategy requires considerably more words than does the code to implement it. The disk routines including the bit maps occupy only 1K bytes and require very little processor time. Whether or not such optimization will have significant impact on system performance remains to be seen. Any kind of estimate would require very rigorous analysis of timing and disk traffic as well as assumptions on the job mix. If analysis would indicate that such techniques are of no value in the "typical" student job mix, I am sure I can persuade a professor to assign a disk sort program as a class assignment and then require that it be tested with a 500 block file consisting of 20 character records.

IV. MIPS FILING SYSTEM

Since MIPS is intended to serve a community of inexperienced student users, simplicity was the keyword in development of a filing system. The design attempts to maintain simplicity without compromising secondary goals of speed and flexibility. Implementation code has not been cast in concrete and will continue to change. Consequently, I will concentrate on universal concepts and point out areas of continued development. The description will be incomplete and perhaps even inaccurate in some details. The discussion will be limited to disk files since the initial version of MIPS does not support other file structured devices such as magnetic tape.

A. MIPS File Structure

MIPS employs a three level file organization as diagrammed in figure 6. A core resident file directory entry contains up to four pointers (Disk Block Numbers) to File Control Blocks (FCBs) on disk. The FCBs in turn contain up to 256 pointers to actual file blocks. File blocks contain data only, i.e., no link or identification fields. This structure is consistent with the MIPS view of the disk as a paging device. By removing the link information to File Control Blocks, some disk space will be wasted when an FCB is not full. However, deletion of files and random access to any block in a file are very burdensome chores when the link information is contained in the file blocks. This organization avoids a lot of those problems.

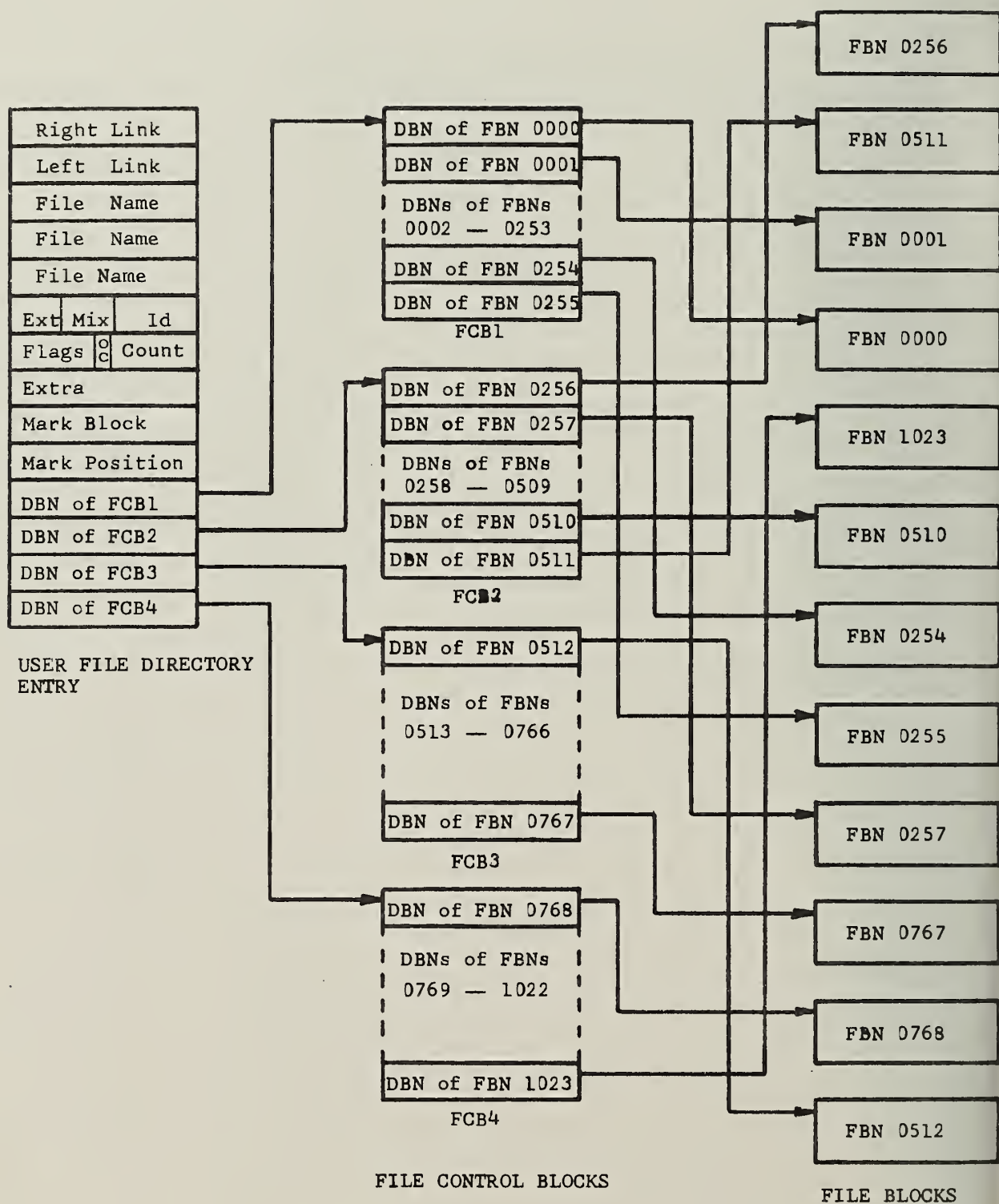


Figure 6. File Structure

File size is limited to 1024 blocks but this is larger than the available disk space and should be sufficient for immediate needs. In terms of card and line images, files containing 10,000 card images (50 character average) or 5,000 line images (100 character average) can be accommodated. Typical user files will require only one FCB and something less than 256 blocks. Reference and access to blocks is by means of a File Block Number (FBN) between 0 and 1023. The user need not be concerned with File Control Blocks, Disk Block Numbers, or actual disk addresses. Management of the File Control Blocks and the mapping of File Block Numbers to Disk Block Numbers are responsibilities of the filing system. The disk service routines in turn convert disk block numbers to actual disk addresses. Finally, files are not limited to contiguously numbered blocks. For example, a file may consist of only two blocks FBN293 and FBN824. In this case, FCB2 and FCB4 are required and each contains only one non-zero pointer.

1. File Windows

The concept of a file window is borrowed from DEC's RSTS monitor [1]. In the MIPS context, a window is an eight word section of a File Control Block. When a file is opened, a buffer is allocated from the pool and linked to the JCB. A window is then loaded into this buffer as part of the open processing. The window permits direct access to eight sequential blocks of a file with only the one disk transfer to read or write the block. An attempt to access the ninth sequential block or any random reference to a block whose DBN is not contained in the resident window will force a disk transfer to bring in a new window.

A resident window number and a window modification bit are maintained in the core buffer along with the window. The modification bit is set whenever

a DBN in the window is modified and indicates to the filing system that the disk copy of that window must be updated before a new window can be loaded. Read transfers do not modify the window. Write/allocate transfers return the DBN of the allocated block to the resident window on completion of the transfer. A window synchronization marker or semaphore prevents the resident window from being written out to disk or overlayed until all disk I/O which would modify it has completed.

The window synchronization marker address is passed to the disk routines and is incremented and decremented according to MIPS marking conventions.

Random access to blocks of a file will therefore require a lot of window I/O in addition to the disk I/O to access blocks. Even at great expense in time, the facility is nice to have under any filing system. Reasonably fast random access to blocks of a file is essential to implement efficient paging of a user program. Paging is discussed in detail in Chapter VII. The more common sequential accessing of files requires very little window I/O and is therefore relatively quick. Some window management functions are required of all file EMTs except .CREATE, .MARKF, and .WAITZ. However, window and FCB management is transparent to the calling task and solely the responsibility of the filing system.

B. Types of Files

Since the paged file structure is flexible enough to handle any type of data, there is no a priori requirement to distinguish between file types. It will be useful, however, to classify files by content and in this way distinguish between four basic types of files as described below:

1. Data Files include pseudo readers, pseudo printers and other listing files, program generated serial

or random access data files, tables, or anything else a user would like to store on disk.

2. Loader Format Files

are generated by the assembler. They contain user object code in the standard absolute loader format which is used on paper tape. This is a convenient representation of user code for transfer to an external medium such as mag tape or paper tape. Loader format files will normally require only one FCB.

3. Program or Object Files

are core images of user code generated by the loader. One FCB is sufficient to map a full 32K virtual address space. Program files are used by the interpreter during execution of a user job.

4. Swap Files

are core image files created by the MIPS swapper whenever a task is removed to disk after exceeding a time slice. Swap files are deleted by the swapper after the task is restarted. One FCB is also sufficient to map any swap file.

C. User File Directory

MIPS does not provide separate directories for system and user files. There is little justification for providing a separate system file directory since the number of system files (assemblers, loader, interpreter, etc.) will be small. Protection mechanisms intended to protect a user's files from other users also serves to protect system files. With very limited disk capacity, it is reasonable to assume that user files will exist only

for the duration of the job. Since few jobs will require more than four files, the number of entries in the UFD will normally be less than thirty even with six jobs in the system. Consequently, the MIPS implementation is very simple. The directory is implemented as a doubly linked list with entries allocated from the buffer pool. Searches are sequential.

The format of a UFD entry is shown in figure 6. The file name is a simple string of ASCII characters. The next word contains a four bit file name extension (valid extensions are listed in Appendix A), the mix index of the job (3 bits), and the Id of the owner (9 bits). Next is a word containing several bits available for use by any future protection mechanisms, an open/closed bit, and a count of the number of blocks in the file. Mark block and mark position fields are used in connection with the .OPENE and .MARKF file EMT's to save the current position in a sequential access file or to contain a start address in loader format and program files. The last four words are the pointers to the File Control Blocks.

D. File Service Requests

MIPS supports standard file operations to create, open, read, write, close, and delete files through simply EMT calls. The filing system code is core resident and reentrant to facilitate service to all operating tasks on a priority basis. All disk I/O is concurrent, i.e., overlapped with processing, interrupt driven, and optimized for minimal disk access time. Figure 7 lists the standard format and required parameters for all file EMT's currently implemented.

One parameter common to all file EMT's is the address of a File Information Block (FIB). The FIB is a 10 word table of parameters peculiar to the file addressed. A description of FIB fields is included in figure 8. The primary protection mechanism is hidden in MIPS treatment of file names.

.CREATE	Create a file.
.WORD FIB	Address of File Information Block.
.OPEN	Open file at block zero.
.WORD FIB	
.OPENE	Open file at last marked block.
.WORD FIB	
.READ	Read from one block of file.
.WORD FIB	
.WORD FBN	File Block Number to be read.
.BYTE SIZE	Number of words to be read (0 = 256 words through 377 = 255 words).
.BYTE START	Start word in block 0-377.
.WORD MARKER	Address of marker which will be incremented on initiation of the transfer and decremented on completion.
.WORD BUFFER	Destination core buffer address.
.WRITE	Write into one block of file.
.WORD FIB	
.WORD FBN	File Block Number to be written.
.BYTE SIZE	Number of words to be written (see .READ).
.BYTE START	Same as in .READ.
.WORD MARKER	Same as in .READ.
.WORD BUFFER	Source core buffer address.
.RLSE	Release one block of file.
.WORD FIB	
.WORD FBN	File Block Number to be released.
.MARKF	Mark current position in file.
.WORD FIB	
.CLOSE	Close file.
.WORD FIB	
.DELETE	Delete entire file.
.WORD FIB	
MOV #MARKER, -(SP)	Wait for completion of transfer. Stack address of marker.
.WAITZ	Wait for marker to go to zero.
BR ERROR	WAITZ returns here if marker will never to to zero.
TST (SP)+	Normal return. Pop wait address.

Figure 7. PAL Formats for File EMTs

File Name	
2nd Char	1st Char
File Name	
4th Char	3rd Char
File Name	
6th Char	5th Char
Slot Index	File Name Extension
Not Used	Error Code
Error Return Address	
EOF Return Address	
Mark Block	
Mark Position	
Not Used	

Six character file name in ASCII.

BYTE 0 One of 16 extensions listed in Appendix A.

BYTE 1 Offset of file slot from top of JCB. Set by filing system.

BYTE 0 Error code returned by filing system. See code listings in Appendix A.

If an error occurs, filing system returns to the caller at this address.

File EMT's return to this address on end of file.

See .MARKF and .OPENE file EMT's.

See .MARKF and .OPENE file EMT's.

Currently not used.

Figure 8. File Information Block Field Definitions

The mix index of the job and the user Id (both extracted from the Job Control Block) are appended to the file name extension bits listed in the FIB to form a word (EMI) which is considered an integral part of the file name. When a file is created this EMI word is stored in the UFD entry along with the six character ASCII name. On all subsequent file service requests, a new EMI is constructed from the JCB and FIB. The ASCII file name in the FIB and the new EMI must match the corresponding parameters in the UFD entry if the access is to be permitted. Two identical jobs (same Id, same file names, etc.) could therefore be run simultaneously with no file interference since the mix indexes will always be different.

Each of the file service requests are described below. New concepts are introduced as required to complete the description of the filing system operation. Programming examples and listings of error codes returned by the filing system are included in Appendix A.

1. .CREATE

The creation of a file is a very simple file operation. The filing system will scan the UFD to verify that a file by the same name, extension, mix index, and Id (EMI) does not already exist. If an identical file does exist, the filing system will flag an error and return to the error return address listed in the FIB. Otherwise, the file is created by allocating a buffer from the pool to serve as a UFD entry, copying the six character file name from the FIB, entering an EMI word into the buffer, and linking the buffer into the UFD.

2. .OPEN and .OPENE

At the time of an OPEN request, the UFD is scanned to verify that the file exists and is not already open. If the file exists or is open, the filing system will flag an error and return to the error return address.

Passing these initial checks the open routines will attempt to find an available file slot in the Job Control Block. Seven file slots are provided in the JCB. The first three are reserved for the job's swap file, pseudo reader, and pseudo printer. Four are floating or available for any other files that a task would like to access. A preassigned slot will always be available since the file is closed. However, if the task already has four other files open, all four floating slots will be occupied. MIPS considers this an error and returns to the error return address. Any task of a job is, therefore, limited to having a swap file, a pseudo reader, a pseudo printer, and four other files open at any one time. The limitation is imposed to keep the JCB small and because open files require one additional system buffer as discussed below.

After a file slot has been assigned, the offset in words from the top of the JCB is stored in the FIB and the address of the file's UFD entry is stored in the first word of the file slot. A system buffer is then allocated and the buffer address stored in the second word of the slot. While the file is open this buffer remains linked to the JCB. It is used to contain the eight word file window, the window modification bit, block count, and the window synchronization marker. On an .OPEN request the first eight words of FCBL are loaded into the buffer. On an .OPENE request the window which contains the DBN of the last marked block (see .MARKF EMT and the Mark Block and Mark Position fields in the FIB and UFD entry) is loaded.

.OPEN and .OPENE requests return to the calling task after the proper window has been loaded. Just prior to return, the Mark Block and Mark Position fields in the UFD are copied into the FIB. This permits the current position in a file to be passed along to several tasks of a job. It is important to the operation of pseudo reader and pseudo printer files which are used throughout the life of the job.

3. .READ and .WRITE

The operation of the filing system routines to read and write file blocks is identical except in the handling of file windows and FCBs, and, of course, in the direction of the transfer. .READ and .WRITE share a common preprocessor with .RLSE, .MARKF, and .CLOSE all of which require the file to be open. The preprocessor picks up the address of the FIB following the call and calculates the JCB slot address from the FIB slot index field. If the file is indeed open, the address of its UFD entry will be in the file slot. An EMI word is constructed and the EMI and file name in the FIB are compared with the corresponding parameters in the UFD entry. If the calling task has destroyed either the slot index, the file name, or the extension in his FIB, or if the file is really not open, these comparisons will fail and the preprocessor will flag a file error.

After preprocessor checks, .READ and .WRITE diverge to handle setup of the transfer. Both routines initially pick up the File Block Number (FBN) parameter following the call and determine if the proper window is resident. If not, window service routines are called to write out the resident window (if it has been modified) and to bring in the window which contains the Disk Block Number (DBN) for this FBN. .READ requires existence of the FCB which contains the correct window and, furthermore, the DBN for this FBN in the correct window must be non-zero. This is just a roundabout way of saying that the addressed block must exist. .READ will return to the End of File address if these conditions are not met. .WRITE will create and clear new FCBs on disk as required. If the addressed block does exist, however, further checking is required to determine if this is a full or partial block transfer. On full block transfers .WRITE will deallocate the old block, clear the DBN

in the window, and then perform a write/allocate transfer. On partial block writes, only a portion of the old block will be overlayed so it cannot be deallocated. In this case, deallocation of the old block and allocation of a new block is not required but the operation is actually slower than a write/allocate transfer. If a new block is created, .WRITE will set the window modification bit and pass the address of the window sync marker to the disk routines as discussed in the .OPEN section.

Read and write routines both check the size and start parameters to avoid block overruns. The sum of these two bytes must not exceed 256. Both routines will flag an error if this condition occurs. The two routines finally reconverge to construct a disk parameter block from the appropriate request parameters and call the disk queuer to queue the transfer.

We should note that waits are implied in the disk transfers associated with windows. Even though the read and write routines do not return to the calling task until any window I/O is complete, other tasks in the system can capitalize on this wait time. .READ and .WRITE do return to the calling task before the primary disk transfer, i.e., that associated with the addressed block, is complete. The caller may therefore continue processing while his disk transfer occurs.

4. .RLSE

The operation of the block release file service request is very similar to .READ and .WRITE. The common preprocessor determines if the file is open, does a file name comparison, and flags any errors encountered. Window checks identical to those in .READ are performed and an error is flagged if the block to be released does not exist. Once the proper window has been brought into core, the disk block is deallocated, the DBN for the addressed FBN is cleared,

and the window modification bit is set. The only disk I/O required is that associated with windows. Since blocks are normally released after a read, the proper window will be resident and no disk I/O will be required.

5. .MARKF

This file request merely copies the Mark Block and Mark Position fields from the caller's FIB into the UFD entry for the file. The only requirement is that the file be open. No disk I/O is required.

.MARKF, .OPEN, and .OPENE work together to pass information along to the various tasks of a job which operate on the same files. For instance, the current block and current byte in the block must be passed on to successive users of pseudo reader and pseudo printer files so that they might open the file at the last marked block (.OPENE) and pick up where the last task left off. Only the .OPENE routine actually operates on the content of these mark fields so the feature can be used to transmit other information also. The mark fields are used by the assembler, loader, and interpreter to pass along the start address of a user program. As was mentioned earlier, the open routines perform the complementary operation of .MARKF by copying the mark fields from the UFD entry into the FIB.

6. .CLOSE

The request to close a file is complementary to the operation of .OPEN. Since the file must be open, the common preprocessor does the initial checks outlined in section 3. The resident window is then written out to disk if it has been modified and the system buffer which held the window is returned to the pool. The open/closed bit in the file's UFD entry is cleared and, finally, the JCB file slot is made available by clearing the two pointer fields.

7. .DELETE

The paged file structure and the availability of several large (256 word) system blocks permit very rapid file deletions under the MIPS filing system. The file must be closed and a UFD scan must be performed to locate the entry pertaining to the file. If the file does not exist, delete will flag an error and return to the error return address. If the file is found, a system block is allocated and one full FCB is loaded into the system block. The FCB is then scanned and all blocks managed by that FCB (DBNs $\neq$ 0) are deallocated. Since a file may have as many as four FCBs, this procedure may be repeated up to three additional times. The time required to deallocate a block is approximately 75 usecs and disk access time to bring in an FCB averages 17 milliseconds. A 1000 block file can therefore be deleted in approximately $4 \times 17 + 1000 \times .075 = 143$ milliseconds. Deletion of the same file in a linked file structure would require 17 seconds in disk access time alone. The speed of file deletions will tend to offset much of the time lost in disk I/O associated with window and FCB management.

8. .WAITZ

The .WAITZ file service request is used in conjunction with .READ and .WRITE marker parameters to wait for completion of disk transfers. Recall that markers are incremented by the disk routines when a transfer is queued and decremented on completion. .WAITZ merely waits for a marker to go to zero. It is therefore possible to wait for several transfers to complete if they all reference the same marker.

The calling task must stack the marker address before the EMT call. On entry .WAITZ tests the marker and returns immediately if it has gone to zero. Otherwise the task is put to sleep (but not swapped out to disk) and will be awakened by the Dispatcher when the marker becomes zero. This is preferable to the calling task wasting CPU time in a test loop.

There is potential danger to the system if the calling task stacks an address which will never go to zero. The system can detect this situation however by merely waiting for all the job's disk I/O to complete. If by that time the marker is not zero, then it will never happen, and .WAITZ will return to the instruction following the call. Since this is an error, a branch to an error handling procedure is usually coded following the call. The normal return, when the marker goes to zero, is to the instruction following the error branch.

V. MIPS STRUCTURE AND OPERATION

This chapter presents a comprehensive discussion of MIPS structure and operation in terms of the work which must be performed. Four system tasks and two user tasks may run concurrently and independently under MIPS supervision. The first two sections of this chapter present detailed discussions of the purpose and operation of each of these tasks. The last section addresses the problem of task management in the multi-task environment.

A. System Tasks

The code required to spool card decks to disk, spool listings to a line printer, swap user tasks to or from core memory, and provide for system communication is implemented in the form of four system tasks. Each of these tasks run as dedicated programs which call upon the MIPS filing system, EMT routines, and other global system subroutines to perform essentially independent functions.

1. Card Reader Service Routines

The system task "CARD" is responsible for resource allocation and initiation of all incoming jobs. While a deck is being read, CARD functions as both a slave and a master to the card reader interrupt service routine. As master it enables interrupts and issues start up commands whenever the reader has stopped for any reason. As slave, it provides buffer service during normal reader operation and error recovery services when read checks or other error conditions occur.

Actual reading of cards is handled by the reader interrupt routine.

Reading is double buffered using two 256 character buffers. When a buffer is filled, the interrupt routine raises one of two buffer request flags, schedules CARD to write the buffer out to disk, and continues reading. At 600 cpm a single buffer will fill in approximately 350 msec. This provides ample time for CARD to write the buffer out to disk before it is needed again even under very heavy disk traffic from other system and user tasks.

Providing no errors occur, the interrupt routine will issue the command to read the next card as soon as the previous card clears the read station. \$ cards will force a stop to allow CARD to separate batched decks. Read checks, timing errors, and hopper decks also force the reader to stop. CARD is then responsible for printing appropriate console messages and adjusting buffer pointers as required to recover from the error condition.

The interrupt service routine supports standard EBCDIC punched cards and binary cards generated by the 360 cross assembler. EBCDIC characters are converted to ASCII at the time of the column interrupt. Two data registers are provided by the CR11 card reader interface. One contains the standard 12 bits of data from one card column. The other is an 8 bit encoding of the 12 bit column data. The 8 bit representation is used as an index into a table of 256 ASCII characters. Translation to ASCII therefore requires only one instruction to move the character from the table. Illegal characters are converted to question marks. To conserve disk space, runs of blanks are reduced to a single byte negative count of the number of blanks in the run. Binary cards contain one byte per column. For no apparent reason, these 8 bits are punched in the 12, 11, 0, ..., 5 rows of the card so a right shift of four is required to make it look like a byte. A mode change card (7-8-9 multi-punch in column 1) is used to switch to binary mode. \$ cards always force the mode back to ASCII.

When a \$JOB card is encountered in the input stream, CARD verifies that the previous job has been initialized. If a \$END card was missing from the previous deck, this essential bit of housekeeping would have been neglected and must be done before the new job can be processed.

One of the \$JOB card parameters is the user's disk claim. Since disk space and a Job Control Block are reserved before the deck is read, CARD must interpret the job card parameters to determine if there is sufficient disk space available to run the job. If either a JCB or sufficient disk space is not available, reading is delayed and a watchdog timer is employed to check available resources once per second. When resources become available they are reserved, and CARD proceeds to create the pseudo reader and pseudo printer files for the job. A non-resident routine is then called to generate a burst page as the first block of the printer file. The job may then continue reading until its \$END or another \$JOB card is encountered.

Job initiation is a very simple matter. After the last card of the deck has been read, the last block of the job's pseudo reader is written out to disk and the file is closed. The job status indicator in the JCB is set to "ready," the address of the EXIT routine is stored as the job's current PC, the job's stack pointer is set to a rather arbitrary address in the EXEC partition, and the JCB is linked to the bottom of the EXEC queue. When the JCB rises to the top of the queue, the job will start up in the EXIT routine which is responsible for sequencing, loading, and initiation of all tasks of a job. Having initialized a job, CARD is then free to continue spooling decks as long as resources are available.

2. Line Printer Service Routines

At end of job, the EXIT routines delete all files belonging to the job except the pseudo printer and other listable output files. The disk blocks

deallocated are immediately made available for use by incoming jobs. The JCB is then linked to the printer queue and the system task "LINE" is scheduled to print the remaining files.

To locate printer files, LINE performs a scan of the user file directory. The pseudo printer file is listed first followed by any other listing files. As blocks are printed they are released and added to the pool of available blocks. On end of file, a listing file is deleted thereby freeing the FCBs and the UFD entry. A count of the number of disk blocks currently in use by a job is maintained in the JCB. This count must go to zero by the time all files have been printed or the system halts. This is one of several such checks which signal system errors, i.e., remaining bugs. After the last file has been printed, LINE calls a non-resident routine to print job statistics, clears the JCB, and links it to the reader queue for reassignment to a new job.

LINE provides buffer and error recover services for the line printer interrupt routine in much the same way that CARD provides these services to the reader interrupt routine. Printers are much simpler devices, however, so the routines to control them are correspondingly simple. The only errors which occur are the printer going offline or running out of paper. When these conditions occur LINE will print a console message and invoke a watchdog timer to check printer status once per second until it is switched back online.

Printing is double buffered again using two 256 character buffers. In a manner analogous to the reader interrupt routine, the printer interrupt routine raises one of two buffer request flags when a buffer has been printed, schedules LINE to reload the buffer, and continues printing. Unlike the card service routines, the printer will stop and wait for a buffer to come in if

required. This should not be necessary, however, since the maximum character transfer rate to this printer is only 560 characters/second (burst rate is much faster but 24 character bursts are followed by 40 msec gaps). There should be no problem running the printer at full rated speed even under heavy system load.

3. The MIPS Swapper

If MIPS is to provide rapid turnaround for small jobs requiring only a few seconds of CPU time, a swapping mechanism must be employed to prevent long jobs from tying up the CPU and memory for extended periods of time. The FILE partition is primarily intended for tasks which communicate with external devices such as a magnetic tape unit or a paper tape reader. Since file transfers to or from such devices require reservation of the device for the duration of the operation, FILE tasks will not be swapped. Tasks running in the EXEC partition, e.g., assemblers, interpreters, etc., may not communicate with reserved devices and may require the services of the CPU and the use of core memory for extended periods of time. EXEC tasks will, therefore, be swapped out to disk at the end of a preset time slice. The time slice will apply to the job rather than to individual tasks and will be initially set to 30 seconds, i.e., long enough to permit short jobs running only in the EXEC partition to complete all processing without being swapped.

The system task "SWAP" is responsible for all operations related to swapping tasks into or out of the EXEC partition. It is scheduled by the clock routine whenever the time slice counter goes to zero. Before attempting to swap the running task out to disk, SWAP will examine the two status indicators for other jobs on the EXEC queue to determine if any job is awaiting execution. This is a very simple scan since there are few possibilities. The first status

word indicates a job on the queue is either ready for execution, i.e., a new job, or was swapped out to disk at some point in time. In this latter case, the second status word indicates the job status at the time of the swap out. If the status was "wait message" and a message has been linked to the job's message queue, or if the status was anything other than "wait message," the job can be swapped back into core so it is considered ready for execution. If there is no other job ready for execution, the current task is permitted to run for another time slice and no swap occurs.

If a suitable replacement is found, the current task will be swapped out to disk. There are several complications, however, which may delay the operation. The status of the running task will be "non-swappable" during execution of the EXIT routine and any time that a system block is allocated to the task, e.g., during file deletions. Since the EXIT routine handles sequencing between tasks and end of job processing, it would complicate things to attempt a swap at that time. System blocks are scarce resources so it is not desirable to have one allocated to the task while the task is idle on disk. SWAP employs a watchdog timer to reschedule the swap operation one second later if the running task is currently non-swappable. Finally, since disk transfers are not serviced in first in first out order, SWAP must also wait for all disk I/O associated with the job to complete. This is accomplished merely by waiting for a system marker in the JCB to go to zero.

In lieu of reserving disk space specifically for swapping, SWAP will create a file and write the core image of the running task into this file. To determine which core blocks are in use, a five word Real Memory Table (RMT) is maintained in the JCB. The name is borrowed from a table used in the XDS-940 CAL timesharing system but the function is not the same[2]. In MIPS the RMT is a bit map indicating status (in use or free) of each of the 80 possible blocks in user core, i.e., 8K-28K. Since all possible non system core is

mapped and not just the EXEC partition of this implementation, partitions may be added, deleted or moved around in core without modification to the swap routine. To remove a task to disk, SWAP merely scans the RMT until it finds a block in use and writes that block out as the first block of the swap file. Continuing through the RMT each block in use is written out in turn.

Management of the RMT is primarily the responsibility of the EXIT routine which will set appropriate bits when system programs such as the assembler or interpreter are brought into core. Several system programs, notably the interpreter, expands as user program pages are loaded. These routines may directly access the RMT to set bits as new core blocks are put to use. The EXIT routine will clear the RMT on completion of a task.

A brief example will be used to summarize the swap process. Consider an EXEC task running on a 20K system consisting of an 8K system area, a 4K FILE partition, and an 8K EXEC partition as diagramed in figure 9a. At the end of a time slice, the task is using only 14 of the 32 blocks in the EXEC partition. If the task has been diligent in maintaining the RMT, it will have bits set as shown in figure 9b. As SWAP scans the RMT the first block in use spans core addresses 60000-60777₈. This block is written out to disk as the first block of the swap file. Continuing the RMT scan, a total of 14 blocks are swapped out to disk. The resulting swap file, as shown in figure 9c, is then closed and the partition may be reloaded. The JCB remains in system core and the RMT is unchanged until the job is brought back into core. The swap in process is naturally the reverse of a swap out. The RMT is again scanned and for each one bit encountered, a block is read from the swap file. The position of a one bit in the RMT determines the core address for the disk transfer. When all blocks have been reloaded, the swap file is closed and deleted.

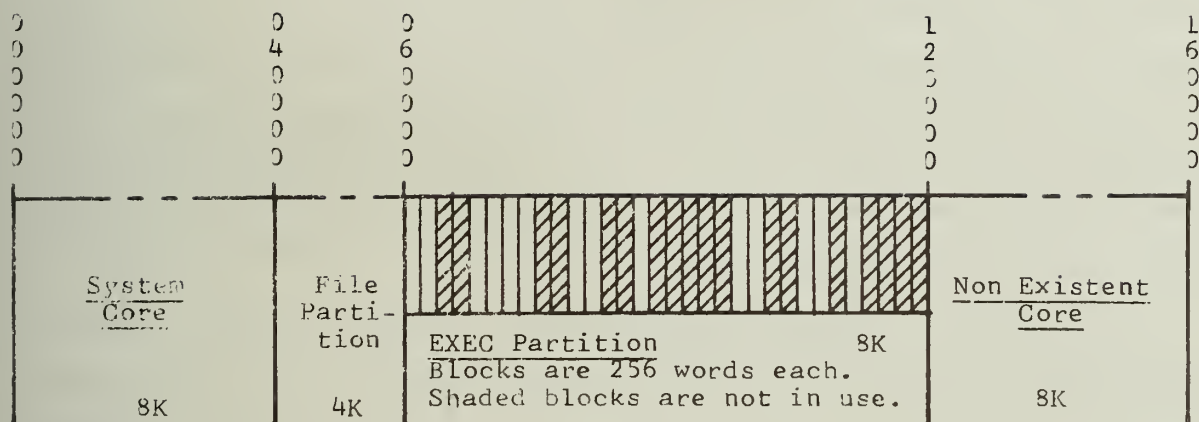


Figure 9a. Core Utilization at Time of Swap

Core address																Swap file blocks for core address:		FBN
File Partition																057777-040000	060000-061000	00
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12K-8K	061000-062000	01
0	1	0	0	1	1	0	0	1	1	1	1	0	0	1	1	077777-060000	064000-065000	02
0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	0	16K-12K	065000-066000	03
Non Existent Core																117777-100000	066000-067000	04
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	20K-16K	067000-070000	05
Non Existent Core																137777-120000	072000-073000	06
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	24K-20K	073000-074000	07
Non Existent Core																157777-140000	076000-077000	08
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	28K-24K	104000-105000	09
																105000-106000	10	
																110000-111000	11	
																111000-112000	12	
																113000-114000	13	

Figure 9b. Real Memory Table
at Time of Swap

Figure 9b. Real Memory Table
at Time of SwapFigure 9c. Resulting
Swap File

4. Console Service Routines and System Communication

The burden of system communication is shared by the system task "CNSOLE," the console TTY interrupt service routines, and two EMT routines: .TYPE and .WAITM. By including in this section discussions of the two EMT routines in relation to the CNSOLE system task, we can address the whole problem of communication between system routines or jobs running under MIPS supervision and an operator at the console TTY.

MIPS routines, systems programs running in either partition, and user programs running under the interpreter may send messages to the console TTY through calls to .TYPE. The message address is passed as the word immediately following the call. For identification purposes .TYPE will insert the caller's mix index and Id number into the message in standard console message format if the first character of the message is a dollar sign (\$). .TYPE will move up to 72 characters into system buffers, link the buffers to the log queue, and schedule CNSOLE to request that the message be printed. The calling routine may continue processing after the message has been queued.

CNSOLE controls printing of the log, manages the TTY interrupt routines, acknowledges requests for use of the keyboard, and responds to valid operator commands. We will delay discussion of commands for a moment and consider the problem of printing the log. Recall that .TYPE schedules CNSOLE, i.e., requests that it be run, whenever a message is queued. When activated by the Dispatcher, CNSOLE checks the status of the console TTY. If it is busy, i.e., a message is being printed or the keyboard is open for input, CNSOLE need not do anything since it will be rescheduled by one of the interrupt routines when the TTY becomes free. If the console is idle, CNSOLE examines the log queue and, finding it not empty, passes the address of the message buffer on top of the queue to the TTY printer interrupt routine. It then enables printer interrupts and returns to the dispatcher which will activate some other task.

Printing of messages is handled completely by the TTY printer interrupt routine. Since messages may be up to 72 characters in length, they may extend over as many as three system buffers. Necessary code to delink buffers from the log queue and return them to the buffer pool is included in the interrupt routine. Typing continues over buffer boundaries until a message delimiter (an ASCII ETX character) is encountered. Printer interrupts are then disabled and CNSOLE is rescheduled. The interrupt routine does not continue on to the next message since CNSOLE may want to acknowledge a keyboard request before continuing with log printouts. A sample console log is included as figure 10 at the end of this section.

An operator may request use of the keyboard for command input by typing a control C. Since the console TTY operates in full duplex mode and keyboard interrupts are always enabled, the keyboard interrupt routine may receive input even if a log message is being printed. The control C causes the interrupt routine to schedule CNSOLE to acknowledge the request. If a log message is being printed, the request is held until the current message has been typed. CNSOLE then acknowledges the request by discontinuing log printouts, typing an acknowledge character (?), and opening the keyboard for input. CNSOLE then returns to the dispatcher so that other tasks may use the CPU while the keyboard interrupt routine accepts the command string. CNSOLE is again scheduled to interpret the command when the operator types a terminating carriage return.

CNSOLE's command string interpreter distinguishes between two types of operator commands: those which refer to a specific job in the system and those which do not. The first category is intended to permit some manual control over the job mix. Directives to abort, stop, or restart a job will be included. In order to respond to directives of this type, the command

string interpreter must first verify that the referenced job does exist. The command is ignored if the job does not exist. At the time of this writing, the CSI recognizes commands of this type and verifies that the job exists, but command service routines are not yet implemented. The current implementation does support several directives in the second category. Non-resident routines are included to set the system clock and to print the job mix. Several small resident routines are currently used to set or print the date or to print the time. Directives to stop the card reader or line printer temporarily, to restart these devices, and to terminate printing of single files are included in the valid command tables but the service routines have not been implemented. Valid command strings are included in the sample log of figure 10. Commands for which service routines do not exist are noted with an asterisk.

The command string interpreter also allows an operator to send a message to any job in the system. Messages may be any arbitrary string of up to 72 characters delimited by single quotes. The CSI first verifies that the addressed job does exist. It then calls a non-resident routine to move the message characters into system buffers and to link the buffers to the job's message queue. Since the message queue list head is in the JCB and messages are stored in resident system buffers, messages may be sent to a job even if the job is not in core. Any number of messages may be queued pending acceptance by the addressed job.

The EMT routine .WAITM delivers console messages to a calling task. If the job's message queue is empty at the time of the call and the task is running in the EXEC partition, .WAITM will schedule SWAP to remove the task to disk. It will not resume until a message appears on the job's message queue. Tasks running in the FILE partition are merely put to sleep but are not swapped out to disk. When a message appears on the queue and the task

is reactivated, or if a message was already on the queue at the time of the call, .WAITM transfers up to 72 characters to the caller's buffer. The buffer addressed is passed as a parameter in the word following the EMT call. Messages are delivered to the calling task in the order sent from the console, one message per call to .WAITM.

The communication features described permit interaction between an operator and a running program. An example of a simple interaction is shown in lines 12, 14 and 16 of figure 10. A program running in the FILE partition requires a tape to be mounted. The routine calls .TYPE to send a message to the console requesting the tape. The routine then calls .WAITM to wait for a console reply. The task continues processing after sending a message indicating that it understood the reply.

The communication methods employed in MIPS reflect to a small degree the thinking of the designers of the RC4000 monitor[3]. Since the current implementation does not permit parallel co-operating tasks, communication facilities need not be quite as general as in that system. Some means by which one task of a job could send a message to a subsequent task would have value even in MIPS, however. This capability would be a simple addition to the existing system.

B. User Tasks

The current MIPS implementation does not permit direct execution of user programs. All user tasks therefore involve execution of a system program in one of the two partitions. User tasks which require execution of a program in the FILE partition are further qualified as FILE tasks. Similarly EXEC tasks are user tasks which require execution of a system program in the EXEC partition. FILE and EXEC tasks run concurrently with the system tasks described in the previous section.

1.	?TIME=14:53:00	Set system clock.
2.	?DATE=07/01/72	Set system date.
3.	2313 BEGIN JOB	System log message.
4.	4765 END OF JOB	System log message.
5.	?MIXX	Command to print job mix.
		MIPS response to MIXX command:
6.	5432 READING	Job #5432 currently reading in;
7.	4000 FREE	JCB #4 is not assigned;
8.	2313 PRINTING	Job #2313 currently printing;
9.	6125 PRINT QUE	Job #6125 awaiting print;
10.	1234 FILE	Job #1234 is running in FILE partition;
11.	3127 EXEC	Job #3127 is running in EXEC partition (end of MIXX response).
		Message from Job #1234.
12.	1234 MOUNT TAPE #145	Log message.
13.	2313 END OF JOB	Log message.
14.	?1234 'TAPE 145 ON UNIT 0'	Console message to Job #1234.
15.	4767 BEGIN JOB	Log message.
16.	1234 THANKS	Message from Job #1234.
17.	?TIME	Command to print time.
18.	6125 END OF JOB	Log message.
19.	TIME=15:02:37	Response to TIME command.
20.	?STCR*	Command to stop card reader.
21.	1234 END OF JOB	Log message.
22.	?DATE	Command to print date.
23.	DATE=07/01/72	Response to DATE command.
24.	2575 BEGIN JOB	Log message.
25.	?GOCR*	Command to restart card reader.
26.	?3127 ABORT*	Command to abort Job #3127.
27.	?MOXY	Illegal command.
28.	HUH?	Response to illegal command.
29.	?MIXX	Another MIXX query.
30.	2575 READING	Job #2575 currently reading in;
31.	6000 FREE	JCB #6 is not assigned;
32.	1000 FREE	JCB #1 is not assigned;
33.	3127 PRINTING	Aborted Job #3127 is printing;
34.	5432 EXEC	Job #5432 is running in EXEC partition;
35.	4767 EXEC QUE	Job #4767 is waiting for EXEC partition;
36.	?STLP*	Command to stop printer.
37.	?KILP*	Command to terminate current listing.
38.	?GOLP*	Command to resume printing.
39.	?KILP*	Command to terminate another listing.
40.	3127 END OF JOB	Log message.
41.	?5432 STOP	Command to stop Job #5432 temporarily.
42.	?5432 GO etc.	Command to restart Job #5432.

Figure 10. Sample Log and Command Strings

1. FILE Tasks

In the current implementation there are no system programs which run in the FILE partition, and, hence, there are no FILE tasks. The author, however, envisions a comprehensive file package to run in this partition. FILE tasks would initially be used to transfer user files between disk and a permanent storage device such as magnetic tape. Any system program which can operate in 4K words of memory can be forced to run as a FILE task so the possibilities are not limited to routines which communicate with external devices. A 4K assembler, for instance, could run in the FILE partition as a FILE task to relieve the load on the EXEC partition. In perhaps a second version of MIPS, if relocatable system routines were used and memory were reconfigured to permit two 8K partitions, file operations might be permitted to run in either partition. If such a reorganization is performed, the concept of a FILE task would essentially disappear. This would permit full utilization of core memory but would add some complexity to task management routines and the EXIT routine which handles some queue management functions and is responsible for loading system programs into core memory.

2. EXEC Tasks

The assembler, loader, and interpreter will all run in the EXEC partition and, hence, the execution of one of these routines is called an EXEC task. Under the current implementation, assemblers, etc. may communicate only with disk and hence, an EXEC task cannot be used to communicate with an external device. We will discuss each of the system programs in the next two chapters so there is no need to go into detail in this section.

C. Task Management

Ideally operating systems are designed on paper and then implemented in code. In practice, however, many details of system design are deferred to the coding phase of the project. As a result, it is difficult to define a precise mechanism for management of tasks until a clear definition of a task develops in the code. While this approach appears a bit shaky in theory, it has proven to be a good approach in MIPS. Design of routines necessary to tie the system together was put off until much of the coding was complete. With other details defined, the problem of CPU allocation essentially dissolved and the implementation of a dispatcher was an easy task.

1. System Priorities

Any discussion of task management routines is ultimately a discussion of system priorities and how these priorities are implemented in code. The MIPS priority structure evolved from the author's discussions with Don Oxley concerning the ETS priority structure[4]. That system in turn was based on the structure used in DEC's RSTS monitor[1] and is similar in concept to a somewhat standard structure used in many real time systems for small machines.

In MIPS we can distinguish between processor (or software), hardware, and dispatcher priorities. The PDP-11 processor operates at any one of eight priority levels (0-7). Hardware device interrupts occur on levels four through seven and will interrupt the CPU if the hardware priority is greater than the current processor priority. Interrupt service routines in MIPS run at a processor priority equivalent to the hardware priority of the interrupting device. Normal nesting of interrupts occurs based on hardware and current processor priorities. The lower four processor priority levels (0-3) are used to implement a priority structure for the various tasks running under

MIPS. The system tasks CARD, LINE, and CONSOLE operate at levels three and two. SWAP is dispatched at priority three but will quickly lower its priority to zero while a swap is in progress. The Dispatcher operates much like any of the system tasks but always remains at priority three. User tasks running in either partition normally run at priority zero and may not raise their priority above two except for very short periods of time through certain supervisor calls. Dispatcher priorities determine which system task is given the CPU if there is more than one task scheduled at one time. Additional comments on dispatcher priorities are included in the next section.

With these numbers in mind, we can consider what happens when an interrupt occurs or a task is scheduled. Recall that tasks are scheduled by the various interrupt service and EMT routines whenever an event occurs which requires reassignment of the CPU. Task scheduling merely requires setting of a single bit in a word called SCHED (scheduler). Peripheral device interrupts, which may occur at any time, simply interrupt any system or user task and return to that task on completion. The return from a clock interrupt is slightly different in that the scheduler word is examined as part of the sequence. If nothing is scheduled, a normal return occurs to the interrupted task. The processor priority at the time of the clock interrupt is examined if any task is scheduled. If the priority of the interrupted task is less than three, it is put to sleep, i.e., its registers are saved, and the Dispatcher is activated. User tasks may, therefore, be interrupted at any time. A system task may be interrupted only if it has lowered its priority below three.

2. The MIPS Dispatcher

The Dispatcher is responsible for assignment of the CPU to system and user tasks. Scheduling of any system task implies a request to run the

Dispatcher as noted above. The Dispatcher itself can also be scheduled by setting a bit in SCHED. This facility is used by the disk interrupt routine when a marker goes to zero since a task may be waiting on the disk transfer just completed. Each of the system and user tasks is represented by one of the six system queues. The queues are ordered and examined according to dispatcher priority which is simply a measure of the importance of the task represented by the particular queue. CARD has timing constraints which require that it have the highest dispatching priority. LINE is next since printing frees up disk space. From highest to lowest priority, tasks are dispatched in the order CARD, LINE, CNSOLE, SWAP, FILE, and EXEC. To find a task which can use the processor, the Dispatcher examines the scheduler word and the status indicators in the JCB on top of each of the system queues.

Dispatcher operation is best explained with the help of the flowchart of figure 11. After some initialization, the Dispatcher first looks at the reader queue. If the status indicator in the JCB on top of this queue indicates that the task is ready, then the system task CARD was interrupted at some previous time and should be reactivated. Registers are immediately restored from the JCB register save area thereby allocating the CPU to the task CARD. The task is also reactivated if it was waiting on an event which has occurred. If a queue is empty or if the JCB status is anything other than "waiting" or "ready," the Dispatcher will examine SCHED to see if the task is scheduled. In the case of the reader queue, a JCB may be on the queue but not assigned to a job. CARD will eventually be scheduled when a new job is placed in the card reader. Similar situations occur with the other tasks. If the task corresponding to a particular queue is scheduled, the Dispatcher merely executes a jump to the appropriate routine. Queues and bits in the scheduler word are examined in turn until the CPU is assigned or all

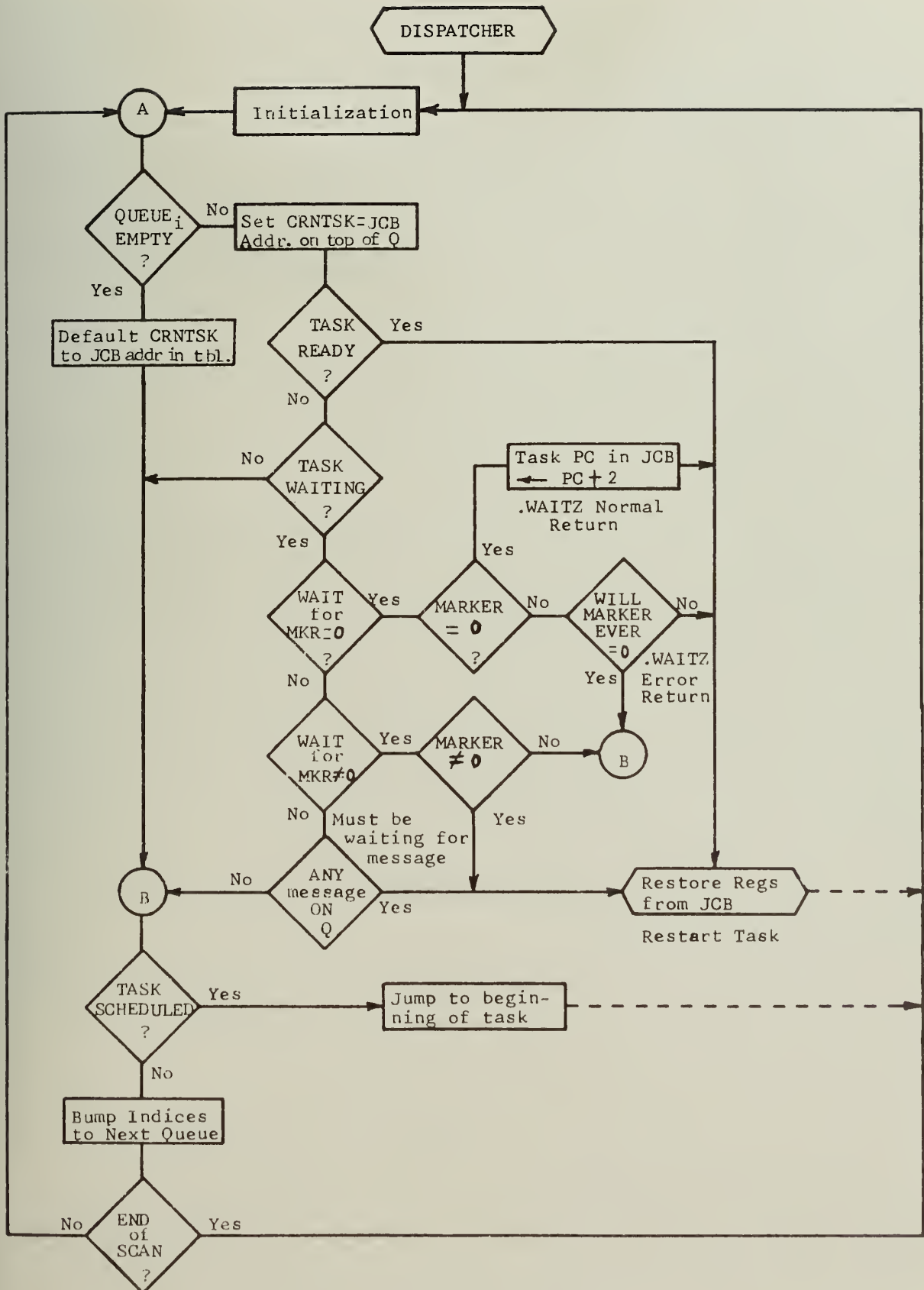


Figure 11. Dispatcher Flowchart

possibilities have been exhausted. If there is no work to be done, the Dispatcher merely loops waiting for conditions to improve.

All system tasks return to the Dispatcher on completion of an operation so that the CPU can be reassigned immediately. When a task of a user job completes, sequencing to the next task is handled by the EXIT routine discussed in the next section and the Dispatcher does not get involved. FILE and EXEC user tasks are never scheduled per se, their JCBs merely appear on the appropriate queues. Queue management is distributed among the system tasks CARD and LINE and the EXIT routine.

3. Sequencing between Tasks of a Job

The EXIT routine handles all system functions necessary to sequence to the next task of a job. System programs running in either partition call EXIT on completion of a task with the .EXIT EMT. To determine which partition is free for use as a scratch area, EXIT will first determine whether the caller was a FILE or EXEC task. It will then open the job's pseudo reader and pseudo printer files and read the last marked block of each into buffers in the free partition. The pseudo reader file is then scanned in search of a \$-card. Any card images flushed are transferred to the printer file with a message saying that they were flushed.

EXIT maintains tables containing the valid names of system programs and flags indicating which partition is required. A program name is scanned off the \$-card and compared with those in the tables. The job is terminated if the \$-card is a \$END card or if it contains an invalid program name. EXIT end-of-job processing routines delete all files belonging to the job except the pseudo printer and other listing files, link the JCB to the printer queue, schedule LINE to print these files, and return control to the Dispatcher for

reassignment of the CPU. If a valid program name is found, EXIT stores the name in the JCB and transfers the rest of the control card to system buffers. System programs such as the assembler or interpreter call .WAITM to deliver the control card to their command string interpreters.

If the next task of a job runs in the same partition as the task which called EXIT, then the system program can be loaded immediately and the task initiated. However, if EXIT was called by a FILE task and the next task runs in the EXEC partition, or vice versa, the JCB must be linked to the appropriate queue to await execution. The JCB status indicator is set to "ready" and the job's program counter is set so that the job resumes execution in the EXIT loader when it bubbles to the top of the new queue.

EXIT contains a small loader responsible for loading system programs from disk into either partition. All system programs exist on disk as core image files. The loader picks up the name of the program to be loaded from the JCB. The file name is the same as that found on the control card. When the file is opened by the loader, the UFD Mark Block and Mark Position fields are transferred to the loader's FIB. For system programs, these words contain the number of blocks to load and the start address of the program. The program is loaded by successive .READ calls to the filing system. On completion of the load, the task is initiated by simply jumping to the start address of the program.

VI. SYSTEM PROGRAMS

With the notable exception of the interpreter, development of system programs to run under MIPS has lagged system development due to a shortage of manpower. The initial version of MIPS requires an assembler, loader, and an interpreter. The design and implementation of the interpreter is the subject of Chapter VII. The current stage of development of the assembler and loader are discussed below.

A. The Assembler

The author is not interested in writing a PAL assembler from scratch to run under MIPS. The intention is to modify an existing ETS[4] PAL-11A absolute assembler to whatever extent is required to make it work with the MIPS filing system. In the current MIPS implementation, there is no requirement that the assembler generate relocatable object modules so the simple PAL-11A should serve the purpose well. The assembler need not know anything about a virtual memory interpreter or the core image form of a user program. It merely generates a loader format disk file containing the user object code in the standard format used on paper tape. At the time of this writing, necessary modifications had just gotten underway and have not progressed much beyond the planning stage. The proposed format for the assembler control card is shown in Appendix B.

B. The MIPS Loader

The loader was written by the author in a very early stage of MIPS development in the hope of gaining some insight into the problems of paging.

The MIPS loader differs from standard loaders in that it does not load a program into core but rather maps a loader format file created by the assembler into a core image file suitable for interpretation. As the loader reads the loader format file, it maps user code into pages in core. Since the partition size is only 8K and the core image form of the user program could be as large as 32K, development of the loader included some experimentation with page replacement policies. The least recently used page replacement policy used in both the loader and in the interpreter evolved after many discussions between the author and Earle Heffley, the fellow who developed the interpreter.

A secondary goal in writing the loader was to develop a standard form for control card command string interpreters (CSI) which would be required in all system programs. A flexible transition matrix parser was developed to perform the CSI functions. The same general form of CSI is being used in both the interpreter and the assembler.

The loader needs some polishing before it would be ready for installation into the system. This would require only a few days work and is not considered a serious problem. The format for the loader control card is also shown in Appendix B.

VII. A VIRTUAL MEMORY INTERPRETER

Operating system development for a PDP-11/20 system is complicated by the fact that the processor does not provide any form of memory protection, relocation, or segmentation hardware. System protection must be implemented in software through interpretation of user programs. While interpretation is a slow process in comparison with actual execution of object code, it does have distinct advantages in the MIPS educational environment. An interpreter can provide extensive diagnostic assistance to students learning to program in assembly language. Furthermore, an interpreter can provide each user with a virtual machine of almost any configuration desired.

The author had a few ideas on a paging interpreter in mind when the project began. Detailed design, development, and coding of the MIPS interpreter was done by Earle Heffley, an undergraduate student in psychology and a very capable programmer. This section is largely a summary of his notes. Development of the interpreter proceeded in parallel with system design and had considerable influence on certain areas such as the disk management routines and filing system.

The MIPS interpreter implements a virtual PDP-11 with as many standard processor features as is practical in a batch environment. A software implemented virtual memory is used to provide the user with a full 32K address space. User code need not be in relocatable form and is not constrained to fit into existing real memory. User programs may also expand dynamically. The interpreter will create additional pages as required.

The MIPS interpreter runs in the 8K EXEC partition. Resident code and tables require less than half of this area and the rest of the partition is used to hold program pages. Eight blocks (2K) are sufficient to run a program

of any size. However, since paging can have considerable impact on program execution time, memory management routines were written to use whatever core is available to increase paging efficiency. In the current implementation, sixteen blocks (4K) of the EXEC partition are available.

A. Paging the User Program

The concept behind the software virtual memory is simple although its implementation is somewhat complex. Basically the interpreter must convert each program address into a real memory location, reading appropriate pages in from disk or possibly creating pages as required. The user program exists as a core image file created by the MIPS loader. This file may contain up to 128 file blocks (pages) of 256 words each, i.e., 32K words. Pages are accessed through standard calls to the MIPS filing system. The basic strategy employed is demand paging although a small amount of lookahead is also performed.

1. Paging When Core is Not Full

All paging operations are performed by one routine called the "Pager". The mechanism is straightforward when the required page is in core or when real memory is not full. The Pager operates on a standard PDP-11 sixteen bit address which is considered a virtual address in the following discussion. The virtual address is divided into page and displacement fields as shown in figure 12. The seven bit page number is actually the File Block Number (FBN) of the page in the core image file. The FBN is used to index into a Core Image Table (CMGTBL) which contains one byte for each of the possible 128 program pages. If the referenced page is in core, this byte will contain a real block number ($\times 2$) which is easily converted to the base address of one of 80 possible blocks of non-system core. The displacement field extracted from the virtual address is added to this base address to give a real memory address.

If the page is not in core, the corresponding byte in the CMGTBL will be zero and the page must be loaded from disk or created. Since we have assumed for the moment that real memory is not full, let us further assume that the Pager can locate an empty block and defer discussion of memory management techniques to a subsequent paragraph. With a block assigned, the Pager need only perform a read to load the required page into core. An attempt to read a non-existent page results in an End of File return from the MIPS filing system. If this occurs, the block assigned is still used as the referenced page, i.e., the Pager has created a new page. Since the assigned block may contain anything, some rather interesting results can occur if the user attempts to read information from a non-existent page. He will, of course, get garbage but this is exactly what would happen if he were running on a real machine.

Since the Pager may be called several times in the interpretation of a single instruction, it is important that operand processing not be held up waiting for a disk transfer. Processing of source and destination operands. is not strictly ordered in the MIPS interpreter and the Pager does not actually wait for a page to come in from disk before returning to its caller. To allow overlap of processing with disk transfers, the Pager maintains a table of 80 markers (MKRTBL), one for each block of non-system core. The address of the marker corresponding to the assigned block is passed to the .READ routine and serves to signal completion of the transfer according to standard MIPS marking conventions. In this way, if processing of one operand is blocked waiting for a disk transfer, the interpreter continues and checks the status of the other operand doing whatever processing it can, including starting other disk transfers, before returning to the former operand.

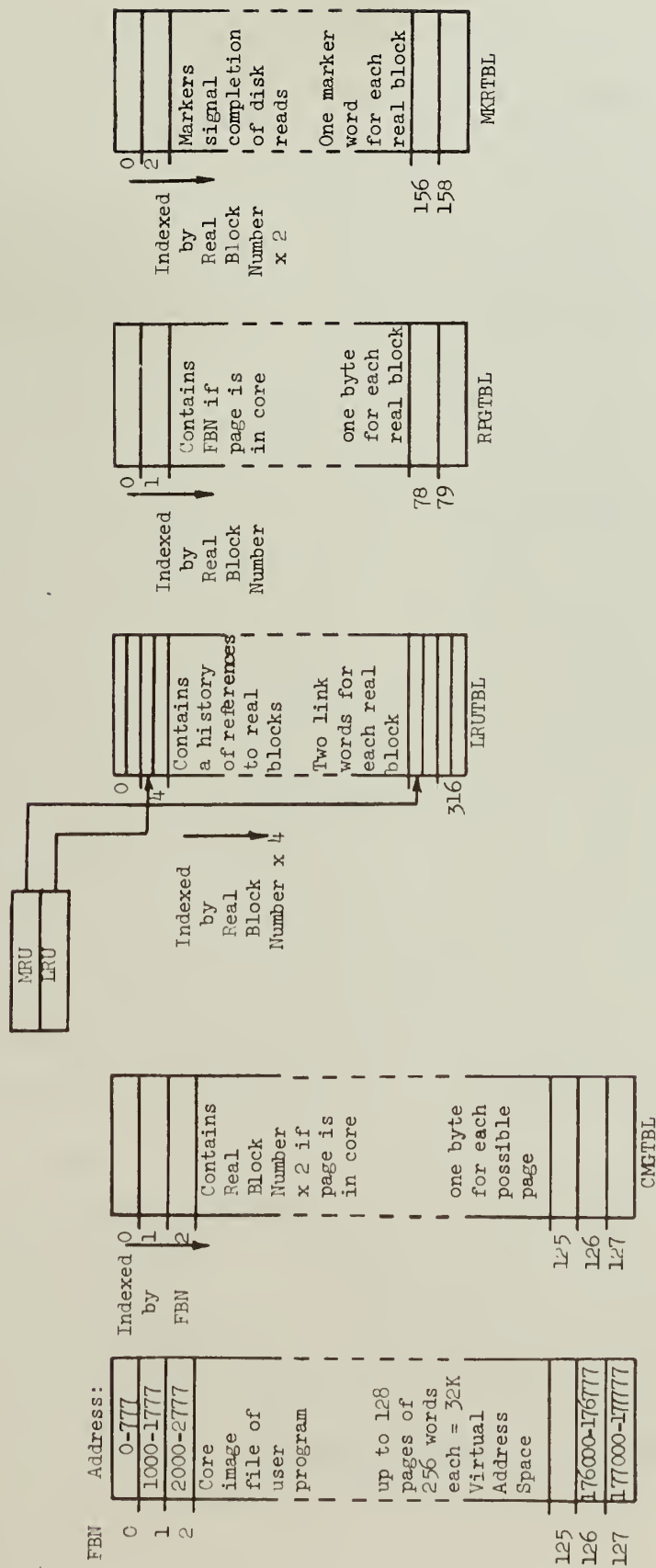
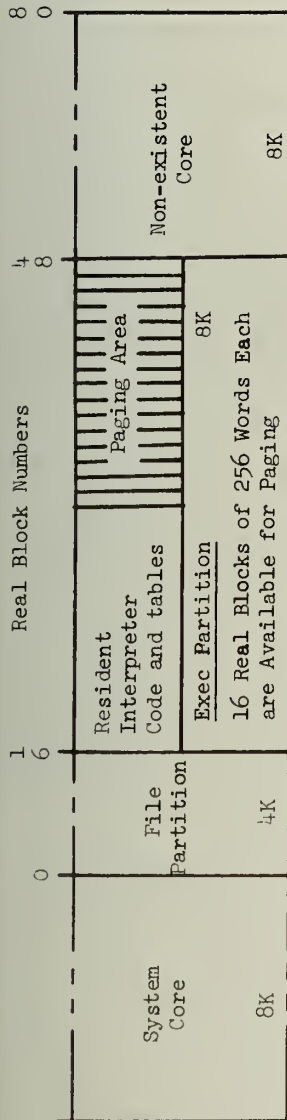


Figure 12. Interpreter Tables

2. Least Recently Used Replacement Policy

When real memory is full and a new page is required, a "least recently used" replacement policy becomes effective. To keep track of which block in core is least recently used (LRU), the Pager maintains the two additional tables diagrammed in figure 12. The LRUTBL is actually a doubly linked list with one node for each of the 80 non-system blocks. The nodes contain only link words and only those blocks currently in use are linked together. A list header contains pointers to the nodes for the most recently used (MRU) and least recently used blocks. There must also be some way to determine which page is currently resident in the LRU block. The FBN for each page in core is saved in the Resident Page Table (RPGTBL) which contains one byte for each of the 80 real blocks.

On each memory reference the FBN extracted from the virtual address is used as an index into the CMGTBL. The CMGTBL contains the real block number which is used to index into the LRUTBL. Links in the LRUTBL are adjusted to make the referenced block most recently used. Linking requires only four instructions so maintenance of the LRUTBL does not add any significant overhead. On small programs which fit into core, the replacement policy has no effect. However, the interpreter does not know the size of the program it is interpreting and, even if it did, it could not assume that the program will not expand at some time. If a swap is required, i.e., a page must be removed to make room for a new page, the real block number of the LRU block can be calculated from the position of the LRU node in the LRUTBL. The real block number is used in turn as an index into the RPGTBL to obtain the File Block Number of the LRU resident page. Once the FBN is known the page can be written out to disk and the new page loaded.

3. Floating Interpreter Pages

To increase paging efficiency three blocks from the available pool are treated in a special manner. The three special blocks, called Floating Interpreter Pages or FIPs, are used to avoid waiting for the write phase of a swap and to decrease execution time required to find an empty page when core is not full. Corresponding to each FIP are two words, one contains its base address and the other serves as a marker for disk writes. When a program page needs to be read in or created, the Pager scans the FIP markers to find one whose contents are not in the process of being written out to disk.

The core block represented by the free FIP is assigned to hold the new page, is linked to the L_RUTBL as the MRU block, and is no longer a floating page. While the disk transfer to load the new page is in progress, the Pager will locate another block to use as a floating page. If core is not full, a free block is located by a simple scan of a bit map similar to the Real Memory Table (RMT) discussed in section V.A.3. If core is full the least recently used block becomes the new FIP and its contents are written out to disk. The FIP is considered busy until its marker goes to zero signaling completion of the disk transfer. Three FIPs are used because the interpreter may have three disk reads queued at one time: one each for source and destination operands, and a third for the page following the one which contains the instruction being interpreted (lookahead).

The sixteen blocks (4K) of the EXEC partition which are available for paging under the current implementation is much larger than most student programs. Consequently, the interpreter will normally demand page the whole program into core and never require another disk access. In such cases, paging overhead is reduced to a small constant which merely reflects the time required to adjust addresses.

B. The Virtual Machine

The MIPS interpreter attempts to offer the user as many of the features of the real machine as is practical and possible. Basically, only five features are denied: access to real device registers, user EMT definition, clock and I/O interrupts, stack overflow error, and processor priority level control. All other standard processor features are implemented on the virtual machine although some are modified slightly.

1. Memory Configuration

As was noted in the discussion on paging, the user has a 32K PDP-11 at his disposal. All addresses can be treated as usable core memory. Trap vectors (except EMT and power fail) in addresses 0-40 have special significance in terms of trap and breakpoint instructions and in the handling of simulated reserved instruction and bus error traps. Interrupt vectors in addresses 40-400 do not exist on the virtual machine and may be used as standard core memory. Similarly the top 4K of address space, which is reserved for device and processor registers on the real machine, may be treated as core since the user is denied access to any real devices and the interpreter maintains a register save area for the user's registers. Finally, the last addressable word (location 177776) is the processor status word on both the real and virtual machines.

2. Instruction Set

All instructions in the PDP-11/20 instruction set are implemented with the following exceptions or modifications:

- a) WAIT and RESET are considered no-ops;
- b) HALT is interpreted as a .EXIT and results in termination of the task;

- c) EMTs are calls to either the MIPS filing system, to other MIPS EMT routines, or to interpreter defined EMT routines.

It should be noted that TRAP, breakpoint, and IOT instructions are implemented and cause the standard processor actions to occur, i.e., the user PC and PS are loaded from the appropriate vector in the user's virtual memory and instruction interpretation continues from the new address.

There is nothing particularly unusual or exciting in the decoding and interpretation of individual instructions in the MIPS interpreter. Instruction decoding simply requires a short series of jumps through several interpreter tables to arrive at the handler for the particular instruction class. The handler, in turn, determines the virtual addresses of any operands and calls the Pager as required to obtain real addresses. The Pager initiates any disk I/O required to bring non-resident pages into core. When the real addresses for all operands are known, i.e., all required pages are in core, the real instruction is either executed or simulated. If the reader is interested, the basic steps in the interpretation of a double operand instruction is summarized in the flow chart of figure 13. Interpretation of single operand instructions, branches, traps, and jumps differ only in the details of the handlers for the specific class.

3. Error Traps

Only the odd address error and reserved instruction trap have any meaning on the virtual machine. Power fail and bus timeout errors do not occur. The stack overflow trap was not implemented in the interpreter because the interrupt vectors do not exist on the virtual machine and, therefore, do not need to be protected.

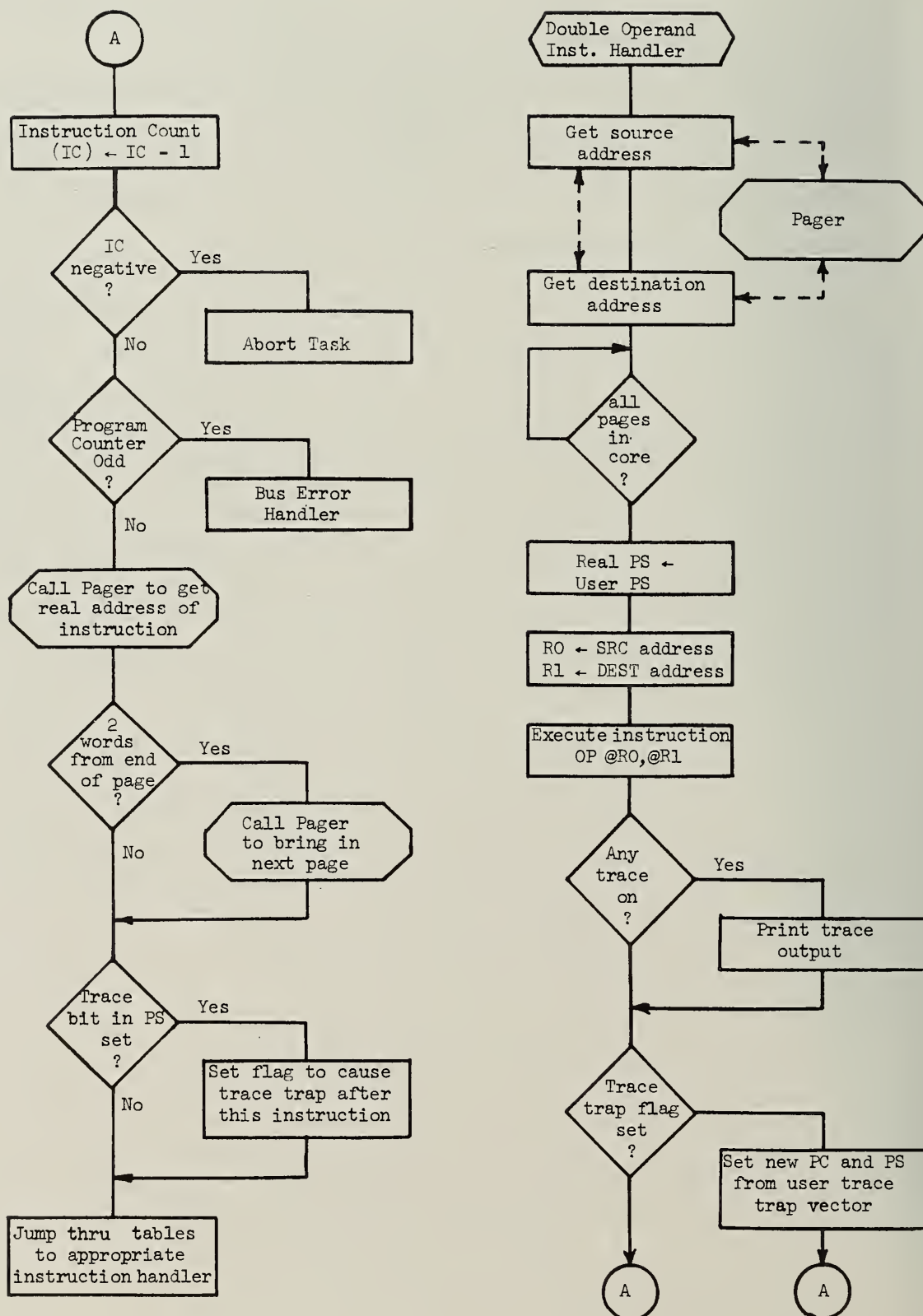


Figure 13. Interpretation of Double Operand Instructions

An odd address error trap occurs when the user attempts to perform any word operation on an odd address. Reserved instruction traps occur if the user tries to execute an illegal instruction. Through interpreter control card (\$EXEC card) options, the user may elect to handle these errors on any of three levels. The first option allows the user to set the trap vectors in his virtual memory and handle all errors with his own error recovery routines. With this option in effect, the interpreter merely loads the user PC and PS from the trap vectors when errors occur. A second alternative is to request the interpreter to handle all errors. The errors then become fatal but an error message specifying the error as completely as possible is printed on the user's pseudo printer file before the interpreter terminates the task. The third option combines the first two. A user may elect to handle errors but also have the benefit of the interpreter error messages.

C. Access to User Files

The interpreter permits access to the MIPS filing system through the standard file EMTs discussed in Chapter IV. In addition to block oriented I/O available through the filing system, the user may also request record oriented I/O through interpreter defined EMTs.

1. Calls to the MIPS Filing System

A user program running under the interpreter has full access to the MIPS filing system. The interpreter, however, intercepts all file EMTs in order to perform error checks and to set up the call in a form that will be understood by the filing system. The user's File Information Block (FIB) and all call parameters must be moved into interpreter scratch areas since the FIB and parameters may involve several program pages. The interpreter also replaces the user's error return address with one of its own in its copy

of the FIB. If an error does occur, e.g., a user attempts to open a file which does not exist, error handling may be on any of the three levels discussed under processor error handling in the previous section. One of three levels is selected by a \$EXEC card parameter.

User file transfers impose an additional burden on the interpreter due to the fact that user buffers may cross page boundaries. Since pages are not contiguous in core, the interpreter uses its own buffers for all disk transfers and then copies the data into the user's buffer. The filing system permits partial block transfers, however, so the current implementation may be changed to permit direct disk transfers to or from user buffers. If a user buffer crosses a page boundary, this would require breaking up a single request into at most two calls to the filing system.

To avoid the added complexity which would be required to synchronize file transfers with the paging mechanism, the next instruction is not interpreted until all disk I/O related to a file service request has completed. Programs running under the interpreter are therefore restricted to non-overlapped I/O. The restriction does not imply any difference in coding of file operations between systems programs running directly under MIPS and programs running under the interpreter. In fact the interpreter is careful to make all file EMTs perform exactly as they would if the program were actually being executed so that systems programs can be debugged under the interpreter prior to installation into the system.

2. Record Oriented I/O

MIPS would impede rather than serve the educational process if students were required to understand the workings of a block oriented filing system before they could see any results from their first machine problem. Many

problems simply require reading of a few data cards and printing the results of some manipulation of that data. To fill the need for some type of simple I/O, three interpreter defined EMTs permit record oriented access to the user's pseudo reader and pseudo printer files. All file operations related to these files, i.e., open, close, read, etc., are handled by the interpreter on behalf of the user. The interpreter buffers all record oriented I/O using 256 word blocks. The PAL call formats for each of the interpreter defined EMTs are summarized in figure 14 at the end of the next section.

a) .GET

The .GET EMT transfers one card image from the pseudo reader to a user defined 80 character buffer. The buffer address is passed as an EMT parameter in the word following the call. Data cards follow the \$EXEC card in the user's deck as shown in Appendix B. The interpreter will continue to transfer card images until the next \$-card is encountered in the pseudo reader file. .GET considers end of file a fatal error, prints a message on the user's pseudo printer, and returns to the system through .EXIT.

b) .GETX

The .GETX EMT is identical to .GET except for a provision which returns control to the user when end of file is encountered on the pseudo reader. The end of file return address is passed as a second EMT parameter. No message is printed.

c) .PUT

Lines may be printed through calls to .PUT. The interpreter will transfer up to 128 characters from a user's line buffer to his pseudo printer file. Line buffers may be any length up to 128 characters and should be terminated with an end-of-transmission (ASCII ETX) character.

D. Diagnostic Tools

The MIPS interpreter provides extensive diagnostic assistance to users. The package includes several levels of aid so that necessary information can be obtained with a minimum of listed output. Diagnostic tools are implemented as versatile dump and trace routines available through simple EMT calls. Each of the routines are discussed briefly below. PAL formats for the EMT calls are summarized in figure 14 at the end of this section.

1. Dump Routines

Three dump routines are included in the interpreter.

a) .RDUMP

Register dumps are requested through calls to .RDUMP. The user's eight registers, his processor status word (PS), and the top three words of his stack are printed in octal on his pseudo printer file.

b) .MDUMP

Memory dumps are initiated through calls to .MDUMP. The EMT instruction is followed by two words containing the low address and the high address of the area to be dumped. The memory dump is preceded by a register dump. Memory contents are then printed in octal, eight words per line followed by sixteen ASCII characters corresponding to the sixteen bytes. Any area of the user's virtual memory may be dumped.

c) .PDUMP

A call to .PDUMP will cause the interpreter to print the addresses of the last ten instructions interpreted. The PC values for the ten instructions are stored in a linked list.

2. Trace Routines

Four levels of tracing are offered to the user to permit efficient debugging. Each of the trace options may be initiated or terminated at the user's discretion through EMT calls.

a) .WTRC

Any word in the user's virtual memory may be monitored through the use of the word trace. The address of the traced word and its old and new contents are printed any time the word is changed. The processor status word may be traced but only changes resulting from instructions referencing the PS as a destination operand will be printed, i.e., normal changes in condition codes will not cause a printout. The word following the EMT call must contain the address of the word to be traced.

Two words may be monitored simultaneously through successive calls to .WTRC. If the user requests that a third word be traced, the first word is no longer monitored. The word trace routine can easily be modified to allow simultaneous tracing of more than two words. Interpretation time will, of course, increase with each addition. Execution of the .WTRX EMT followed by the address of a traced word terminates tracing of that particular location. No action is taken if the addressed word is not being traced at the time of the call to .WTRX.

b) .PCTRC

The interpreter will begin dumping the program counter for each instruction executed after .PCTRC is called. PC values will be printed in octal, ten values per line. This option is turned off by executing the .PCTRX EMT.

c) .BRTRC

It is often desirable to know program flow without having to scan through the addresses of all instructions executed. After the branch trace option is turned on, the old and new contents of the program counter are printed each time "normal" program flow is changed by execution of a JMP, JSR, RTS, TRAP, IOT, breakpoint, or any branch instruction. The branch trace is terminated with the .BRTRX EMT.

d) .TRC

A call to the .TRC routine initiates a full trace option. For each instruction interpreted, the instruction address, instruction mnemonic, operand mnemonics (register mode only) or operand addresses, all words or registers referenced or changed, and the values of the five condition codes are printed. Elusive program bugs are quickly located and corrected when the full trace option is used. Tracing is terminated with a .TRX EMT call.

e) Trace Routine Priority

To eliminate redundant printout, a priority structure exists for the trace routines. PC tracing (.PCTRC) causes branch tracing to be suspended. If the PC trace option is turned off, branch tracing will be reinstated. Similarly full trace (.TRC) will cause all other tracing to cease until .TRX is called. Other combinations of tracing may occur simultaneously.

E. Interpreter JCL and Options

Interpreter options are invoked through parameters on the \$EXEC interpreter control card. Option phrases of the form <keyword>=<parameter>

EMT FORMAT	DESCRIPTION
.GET .WORD Buffer Address	Read data card from pseudo reader into this buffer.
.GETX .WORD Buffer Address .WORD End of File Return Address	Read data card from pseudo reader into this buffer and return to this address on EOF.
.PUT .WORD Buffer Address	Write one line to pseudo printer from this buffer.
.RDUMP	Dump registers.
.MDUMP .WORD Low Address .WORD High Address	Dump area of memory. Start dump at this address. End dump at this address.
.PDUMP	Dump locations of last ten instructions interpreted.
.WTRC .WORD Address of Word to be Traced	Initiate word trace of this location.
.WTRX .WORD Address of Word being Traced	Terminate word trace for this location.
.PCTRC	Initiate Program Counter trace.
.PCTRX	Terminate Program Counter trace.
.BRTRC	Initiate branch trace.
.BRTRX	Terminate branch trace.
.TRC	Initiate full trace.
.TRX	Terminate full trace.

Figure 14. Call Formats for Interpreter Defined EMTs

are translated into interpreter flags and parameters by a Command String Interpreter (CSI). The standard control card format and options are described below.

$$\$EXEC, \left[\text{OBJ}=\langle \text{filename} \rangle, \text{ERR}=\begin{Bmatrix} \text{PGM} \\ \text{INT} \\ \text{FLG} \end{Bmatrix}, \text{EMT}=\begin{Bmatrix} \text{PGM} \\ \text{INT} \\ \text{FLG} \end{Bmatrix}, \text{CNT}=\begin{Bmatrix} \langle \text{octal no.} \rangle \\ \langle \text{decimal no.} \rangle \\ \text{OFF} \end{Bmatrix}, \text{LIN}=\begin{Bmatrix} \langle \text{octal no.} \rangle \\ \langle \text{decimal no.} \rangle \\ \text{OFF} \end{Bmatrix} \right]$$

1. Object File Name

The first option phrase allows the user to name the object file to be interpreted. The $\langle \text{filename} \rangle$ may be any string of six or less alphanumeric characters as long as the first character is alphabetic. If the phrase is missing the CSI defaults the file name in a manner consistent with defaults taken when an object file is created by the MIPS loader.

2. Processor Error Handling Options

The second option selects one of three error handling procedures discussed under Error Traps in section B. The PGM specification allows the user to handle all error traps. Errors are fatal under the INT option but the interpreter will print a diagnostic message. The interpreter will print a diagnostic and perform the error trap sequence if the FLG option is specified. If the phrase is missing, the INT option is taken.

3. File Error Handling Options

Filing system errors may also be handled on any of the three levels listed above. If an error occurs and the user has specified either the PGM or FLG option, the interpreter continues instruction interpretation at the error return address listed in the user's FIB. The default option is again INT and all filing system errors become fatal.

4. Instruction Count

The interpreter decrements an instruction counter for each instruction interpreted to provide infinite loop protection. The "CNT=" may be followed by an octal or decimal number or by the keyword OFF if no instruction count is to be kept. The count is in thousands of instructions to be interpreted and is defaulted to 20000 instructions if the phrase is missing. The task is aborted if the count goes to zero.

5. Line Count

The interpreter also monitors the number of lines printed by the program. All lines printed including .PUT requests, and all dump and trace output is counted against this limit. The default limit will be set to 500 lines. Again interpretation is terminated if the limit is exceeded.

6. Control Card Format Variations

The interpreter CSI is a small transition matrix parser which will accept any command string and try to make sense out of it. Currently option phrases must be in the order shown and keywords should be spelled correctly. Since defaults are taken when phrases are not understood, the only fatal JCL error is a non-existent object file.

VIII. CONCLUSIONS

MIPS development was in the debugging phase when the author terminated the effort to write this paper. Hence, we cannot include an evaluation of system performance. All monitor code described has been implemented but is not fully operational. Task management routines are operating and are capable of running the various tasks without interference. Two simple test programs have been run concurrently in the two partitions. Console service routines including teletype interrupt routines, non-resident segments, and communication EMTs are also functioning properly. Disk management routines were tested with single block transfers but the author has not run a full test of the optimization routines. Card reader and line printer interrupt routines are working but their associated system tasks CARD and LINE have not been fully tested. None of the swapping routines are operational and will require some additional work. Parts of the filing system were operational but some minor changes were being made when the debugging effort was interrupted. The author plans to resume the debugging effort when this paper is complete but it is unlikely that the system can be fully operational before he leaves the University. Approximately one month of full time effort would be required to bring the monitor, assembler, loader, and interpreter up to full operational status.

The system as described should be quite adequate for this application. The author does, however, recognize several areas where continued development would enhance system performance. The installation of system programs will be

done essentially manually for the initial system. Simple changes in the loader could permit dynamic replacement or creation of system routines. A facility to run rather than interpret user programs should be added so that error free programs can capitalize on the full speed and power of the processor. Additional work is required in the areas of protection, sharing, and access to files. The ability to save files on disk should be implemented to satisfy system programmers running stand alone. This addition would require only a very simple modification to the filing system and EXIT routines. Finally a comprehensive file package should be written to run in the file partition to permit file transfers between disk and permanent storage mediums such as paper tape or magnetic tape.

LIST OF REFERENCES

- [1] Teichholtz, N., Program listings of the Resource Timesharing System, RSTS-11, Digital Equipment Corporation.
- [2] Watson, R. W., "Timesharing System Design Concepts," McGraw Hill, 1970.
- [3] Hansen, P. B. (ed.), "RC4000 Software Multiprogramming System," 2nd edition, A/S Regnecentralen, Copenhagen, Denmark, 1971.
- [4] Oxley, D. W., "The Design and Implementation of an Educational Timesharing System for the PDP-11," Master Thesis, University of Illinois, Urbana, June, 1972, Report No. UIUCDCS-R-72-520.

APPENDIX A

FILING SYSTEM ERROR CODES AND PROGRAMMING EXAMPLE

A. Filing System Error Codes

Errors detected by the MIPS filing system force an immediate return to the caller at the error return address listed in his File Information Block (FIB). All error checking is performed before any action is taken which could modify any file. Error codes returned to the FIB are listed below.

EMT FORMAT	FILE STATUS MUST BE	ERROR CODE	ERROR DESCRIPTION
.CREATE .WORD FIB Address	Non-existent	001	File by same name already exists.
.OPEN or .OPENE .WORD FIB Address	CLOSED	021 022 023	File not closed. File does not exist. Four files already open.
.READ .WORD FIB Address .WORD FBN .BYTE Size .BYTE Start Word .WORD Marker Address .WORD Buffer Address	OPEN	041 042 044	File not open. File name error.  Illegal size or start word.
.WRITE .WORD FIB Address .WORD FBN .BYTE Size .BYTE Start Word .WORD Marker Address .WORD Buffer Address	OPEN	061 062 064	File not open. File name error.  Illegal size or start word.
.RLSE .WORD FIB Address .WORD FBN	OPEN	101 102 103	File not open. File name error.  Block does not exist.
.MARKF .WORD FIB Address	OPEN	121 122	File not open. File name error. 
.CLOSE .WORD FIB Address	OPEN	141 142	File not open. File name error. 
.DELETE .WORD FIB Address	CLOSED	161 162	File not closed. File does not exist.

Notes:

 File name or extension has been changed since file was opened.

B. Programming Example

The program listed below demonstrates many features of the MIPS filing system. It is a simple flush program which scans an existing file for the first occurrence of an ASCII dollar sign. Disk I/O is double buffered and scan processing is overlapped with disk transfers. Blocks of the file being scanned are released after they have been examined. If a dollar sign is found, the block in which it appears is not released, the position in the file is saved, and the file is closed. Another task may reopen the file and continue processing of data starting with the \$ character. Typical error handling code is also listed for all possible errors which might be encountered. To promote clarity, the code is neither clever nor optimal. If this program were run under the MIPS interpreter rather than as a system program, the interpreter would not overlap the disk transfers with interpretation of instructions but, otherwise, operation would be identical.

```

;
START:  MOV    #START,SP      ; Set stack pointer.
        .OPENE                ; Open data file at last marked block.
        .WORD DATAFILE      ; Address of File Information Block (FIB).
        MOV    DFFMKB,RO      ; .OPEN and .OPENE copy MARK BLOCK
        MOV    DFFMKP,R1      ; and MARK POSITION fields from UFD to FIB.
        DEC    RO             ; DFGET1 will increment block number before read.
        JSR    PC,DFGET1      ; Read last marked block into buffer 1.
        ADD    #DFBUF1,R1     ; Add buffer address to position in block.
LOOP:   JSR    PC,DFGET2      ; Read next block into buffer 2.
        TST    EOFGL1         ; End of file when trying to read into buffer 1?
        BNE    GOTEOF         ; Yes, go abort the job.
        MOV    #DFMKR1,-(SP)  ; Stack address of marker which will go to zero
        .WAITZ                ; on completion of buffer 1 transfer and wait.
        BR     WAITERR        ; Wait error here would be a system error.
        TST    (SP)+          ; Pop address of wait marker off stack.
;
NLOOP1: CMPB   (R1),#'$        ; Begin scan of buffer 1.
        BEQ    FOUND          ; If dollar sign found, then were done.
        CMPB   (R1)+,#ETX     ; End of file character?
        BEQ    GOTEOF         ; Yes, didn't find dollar sign. Go abort the job.

```

```

CMP    R1,#DFEND1      ; End of buffer 1?
BNE    NLOOP1          ; No, continue scan in buffer 1.
MOV    DFFBN1,RLFBN1   ; Store File Block Number of block just scanned.
.RLSE                      ; Release that block.
.WORD  DATAFILE        ; FIB address.
RLFBN1: .WORD 0          ; FBN to be released.
JSR    PC,DFGET1       ; Read next block into buffer one.
MOV    #DFBUF2,R1      ; Set buffer pointer to top of buffer 2.
TST    EOFLG2          ; End of file when trying to read into buffer 2?
BNE    GOTEOF          ; Yes, go abort the job.
MOV    #DFMKR2,-(SP)   ; Stack address of marker which will go to zero
.WAITZ                  ; on completion of buffer 2 transfer and wait.
BR     WAITERR          ; Wait error here would also be a system error.
TST    (SP)+           ; Pop address of wait marker off stack.
;
NLOOP2: CMPB (R1),#'$    ; Begin scan of buffer 2.
BEQ    FOUND           ; If dollar sign found then were done.
CMPB   (R1)+,#ETX      ; End of file character?
BEQ    GOTEOF          ; Yes, didn't find dollar sign. Go abort job.
CMP    R1,#DFEND2      ; End of buffer 2?
BNE    NLOOP2          ; No, continue scan in buffer 2.
MOV    DFFBN2,RLFBN2   ; Store file block number of block just scanned.
.RLSE                      ; Release that block.
.WORD  DATAFILE        ; FIB address.
RLFBN2: .WORD 0          ; FBN of block to be released.
MOV    #DFBUF1, R1     ; Reset buffer pointer to top of buffer 1.
BR     LOOP            ; Go read next block into buffer 2.
;
DFGET1: INC    RO        ; Block read subroutine for buffer 1.
MOV    RO,DFFBN1       ; Bump the block number prior to read.
.READ                  ; Store FBN of block to be read.
.WORD  DATAFILE        ; Read the block.
; FIB address.
DFFBN1: .WORD 0          ; FBN of block to be read.
.WORD  0                ; Full 256 word block.
.WORD  DFMKR1           ; Address of marker for disk I/O on buffer 1.
.WORD  DFBUF1           ; Address of buffer 1.
RTS    PC              ; Return to caller.
;
DFGET2: INC    RO        ; Block read subroutine for buffer 2.
MOV    RO,DFFBN2       ; Bump the block number prior to read.
.READ                  ; Store FBN of block to be read.
.WORD  DATAFILE        ; Read the block.
; FIB address.
DFFBN2: .WORD 0          ; FBN of block to be read.
.WORD  0                ; Full 256 word block.
.WORD  DFMKR2           ; Address of marker for disk I/O on buffer 2.
.WORD  DFBUF2           ; Address of buffer 2.
RTS    PC              ; Return to caller.
;
DFMKR1: .WORD 0          ; Marker for disk I/O on buffer 1.
DFMKR2: .WORD 0          ; Marker for disk I/O on buffer 2.
;
DTEOF:  CMP    RO,DFFBN1 ; Filing system returns here on End of File.
BNE    SETFL2          ; Find out which buffer was to be loaded.

```

```

SETFL2:  COM  EOF1G1      ; If buffer 1, set End of File flag 1
        RTS   PC          ; and return normally from read.
SETFL2:  COM  EOF1G2      ; If buffer 2, set End of File flag 2
        RTS   PC          ; and return normally from read.
;
FOUND:   DEC   RO         ; When dollar sign is found,
        MOV   RO, DTAMKB  ; store FBN of block where it was found.
        CMP   RO, DFFBN1  ; Was dollar sign found in buffer 1?
        BNE   FINBF2     ; No, must have been buffer 2.
FINBF1:  SUB   #DFBUF1,R1  ; Subtract buffer 1 base address from dollar sign
        BR    FMARK      ; position in buffer 1
FINBF2:  SUB   #DFBUF2,R2  ; or subtract the buffer 2 base address.
        MOV   R1, DTAMKP  ; Store position (byte) of $ in block into FIB.
        .MARKF          ; Mark the current position in file.
        .WORD DATAFILE ; FIB address.
        .CLOSE         ; Close the file.
        .WORD DATAFILE ; FIB address.
        .EXIT          ; Return to monitor.
;
WAITERR: .TYPE          ; Error handling routines.
        .WORD MSG1      ; On a wait error, send message to console.
        MOV   #7777,R1   ; Message address.
        .ABORT          ; Error code into R1.
;
MSG1:    .ASCII/$WAIT ERROR - SYSTEM BUG!//
;
GOTEOF:  MOV   #7776,R1   ; If EOF is encountered before dollar sign,
        .ABORT          ; set error code and abort the job.
;
DTFERR:  MOV   DTACOD,R1  ; On filing system error, move error code to R1.
        BIC   #177600,R1 ; Take out the garbage.
        .ABORT          ; Abort the job.
;
DATAFILE: ; File Information Block (FIB)
;
DTANAM:  .ASCII/MYDATA;   ; Data file name.
DTAEXT:  .BYTE EX.DTA     ; File name extension.
DTANDX:  .BYTE 377        ; File slot index when file is open.
DTACOD:  .BYTE 0          ; Error code returned by filing system.
DTAXTR:  .BYTE 0          ; Extra byte not used.
DTAERA:  .WORD DTFERR     ; Error return address.
DTAEOF:  .WORD DTFEOF     ; End of File return address.
DTAMKB:  .WORD 0          ; Mark Block.
DTAMKP:  .WORD 0          ; Mark Position in block.
DTANOT:  .WORD 0          ; Extra word not used.
;
EOF1G1:  .WORD 0          ; End of File flag for buffer 1.
EOF1G2:  .WORD 0          ; End of File flag for buffer 2.
;
DFBUF1   = .              ; Buffer 1.
        . = .+512.        ; 256 words.
DFEND1   = .              ; End of buffer 1.
DFBUF2   = .              ; Buffer 2.
        . = .+512.        ; 256 words.
DFEND2   = .              ; End of buffer 2.

```

C. File Name Extensions

File name extensions defined under the current MIPS implementation are listed below. The labels will be defined in the assembler's permanent symbol table. The top four bits of the extension byte are combined with the user's mix index and Id to create an EMI word which is considered an integral part of the file name.

<u>Assembly Label</u>	<u>Definition</u>	<u>Type of File</u>
Not defined	000	
EX. CRD	020	Pseudo Reader File
EX. LPT	040	Pseudo Printer File
EX. SWP	060	Swap File
EX. PAL	100	PAL SourceFile
EX. LDA	120	Loader Format File
EX. OBJ	140	Object File (Core Image)
EX. DTA	160	Data File
EX. LST	200	Listing File
Not defined	220	
Not defined	240	
Not defined	260	
Not defined	300	
Not defined	320	
Not defined	340	
EX. SYS	360	System Program

APPENDIX B

CONTROL CARDS AND DECK SETUP

A. Assembler Control Card Format

```
$ASM[,SRC=<filename>,LDA=<filename>,LST=<filename>]
```

B. Loader Control Card Format

```
$LOAD[,OBJ=<filename>,LDA=<filename>,<filename>,...,<filename>]
```

C. Typical Deck Setup

1. Assembly Only

```
$JOB,ID=123,DISK=120
$ASM
  ==
  <PAL Source Deck>
  ==
$END
```

2. Assemble, Load, and Go with no Named Files

```
$JOB,ID=456
$ASM
  ==
  <PAL Source Deck>
  ==
$LOAD
$EXEC
  ==
  <Data Deck>
  ==
$END
```

3. Assemble, Load, and Go with Several Named Files

```
$JOB, ID=124, DISK=425, THIS IS A COMMENT
$ASM, LDA=MYLDA, LST=MYASMB
```

```
==
<PAL Source Deck>
```

```
==
$LOAD, OBJ=MYOBJ, LDA=MYLDA
$EXEC, OBJ=MYOBJ, ERR=PGM, EMT=FLG, CNT=90., LIN=7777
```

4. Load and Go from 360 Object Deck

```
$JOB, ID=517, DISK=900.
$LOAD, LDA=*
```

```
==
<360 Binary Deck>
```

```
==
$EXEC, LIN=999
$END
```


BIBLIOGRAPHIC DATA SET	1. Report No. UIUCDC -R-72-536	2.	3. Recipient's Accession No.
Title and Subtitle The Design and Implementation of a Multi-Task Batch Operating System and Paging Interpreter for the PDP-11			5. Report Date August, 1972
Author(s) James D. Miller			6.
Performing Organization Name and Address Department of Computer Science University of Illinois Urbana, Illinois 61801			8. Performing Organization Rept. No.
Sponsoring Organization Name and Address National Science Foundation Washington, D. C.			10. Project/Task/Work Unit No.
			11. Contract/Grant No. GJ-812
			13. Type of Report & Period Covered Research
			14.
Supplementary Notes			
Abstracts MIPS is a multi-task batch operating system designed to support a community of student users taking introductory courses in assembly language programming. The system features disk optimization routines which minimize disk access time, a general purpose filing system, and an interpreter which pages the user program to implement a 32K virtual machine. MIPS is written in the PAL-11 assembly language for use on the PDP-11/20 computer.			
Key Words and Document Analysis. 17a. Descriptors Batch Operating Systems, Multi-task, Multiprogramming, Mini-computers, Virtual Memory, Paging, Interpreter.			
Identifiers/Open-Ended Terms			
COSATI Field/Group			
Availability Statement Unlimited Release		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
		20. Security Class (This Page) UNCLASSIFIED	22. Price

SEP 22 1972



UNIVERSITY OF ILLINOIS-URBANA



3 0112 002099031